Lab 5: Image Stitching for Panorams

What to Submit

Submit this iPython Notebook--containing all your code for the programming exercises below--on Canvas.

Programming Exercise

For this assignment, you will be writing a program that creates an image panorama from 2 or more images. In general this technique should be applicable to any number of photographs. The approach described below will work well for collective fields of up to 90 or even 120°, but won't produce ideal results for large fields of view approaching or surpassing 180°. For large fields of view, cylindrical or spherical projection is required.

When we construct a panorama, we assume that all of the photographs were taken from the exact same location and that the images are related by pure rotation (no translation of the camera). The easiest way to create the panorama is to project all of the photos onto a plane. One photo must be selected (either manually or by your program) to be the base photo. The other photos are aligned to this base photo by identifying a homography (a planar warp specified by 4 pairs of source/destination points) relating each pair. Each of the other images is appropriately warped and composited onto the plane (the base image doesn't need to be warped).

In describing what you need to do, there will be a running example using the three photos below:

Image 1

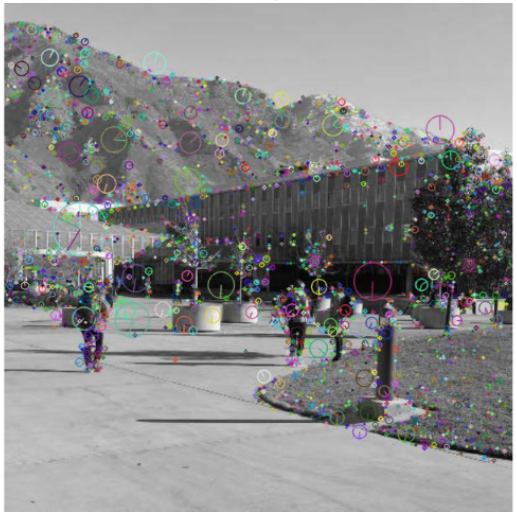
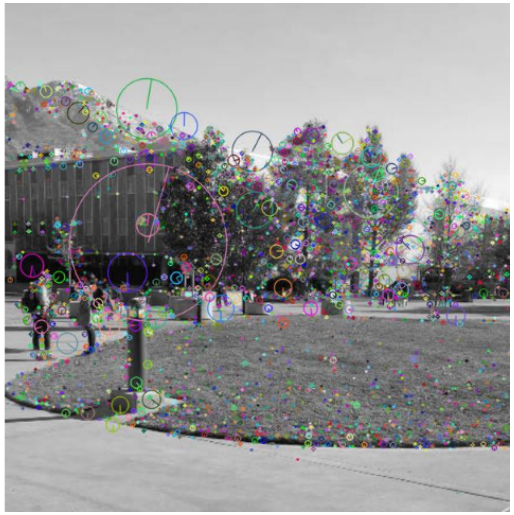
Image 2

Image 3

Part A: Find Interest Points/Descriptors in each Input Image

We will be using OpenCV for this project, which you should already have installed. Although the SIFT algorithm is patented and used to require a separate install, it is now included in the newer versions of OpenCV. A good tutorial on how to use SIFT features in OpenCV is found here (https://docs.opencv.org/trunk/da/df5/tutorial_py_sift_intro.html). The first step to registering or aligning two images is to identify locations in each image that are distinctive or stand out. The `sift.detectAndCompute()` routine produces both these interest points and their corresponding SIFT descriptors. The first step of producing a panorama is to load all of the relevant images and find the interest points and their descriptors.

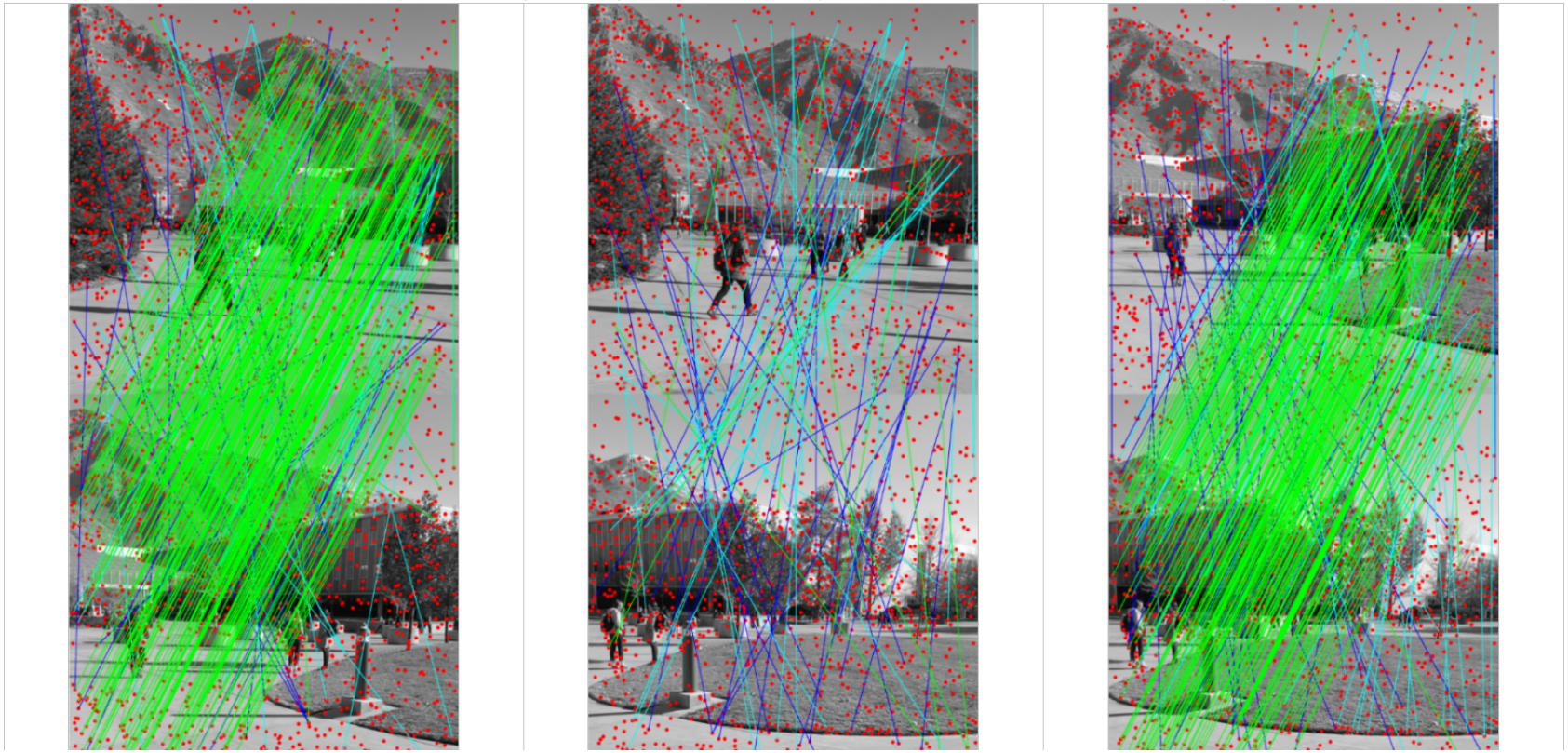
See the circles on each image below indicating the sift keypoints that were found (note that we downsampled the images to 600 x 600 pixels before extracting SIFT). The circles are scaled according to the scale at which each keypoint was detected at and the radius indicates the dominate gradient magnitude. Output a similar visual in your code.

Part B: Matching Features

Next, given the features present in each image, you need to match the features so as to determine corresponding points between adjacent/overlapping images. This page (https://docs.opencv.org/trunk/dc/dc3/tutorial_py_matcher.html) provides details to do feature matching using `cv2.BFMatcher()`, analogous to the approach proposed by David Lowe in his original implementation. Be aware that the resulting match is one directional. You want to find putative pairs--pairs of points which are each other's best match (e.g. there might be 3 points in image I1 for which a point q in image I2 are the best match, only one of these could be the best matching point p in I1 for that point q in I2). In this part you need to compute the set of putative matches between each pair of images.

Look at the pairs of images and the lines showing the estimated matches (putative matches are green lines, one way matches are cyan or blue). Output a similar visual in your code (the images can be side by side, it doesn't have to be vertical).

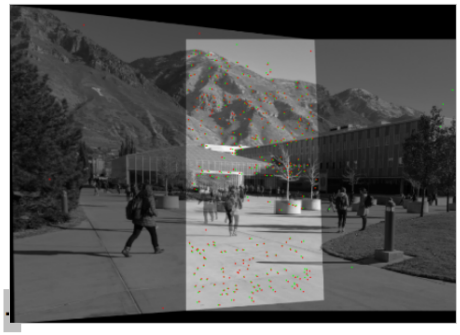
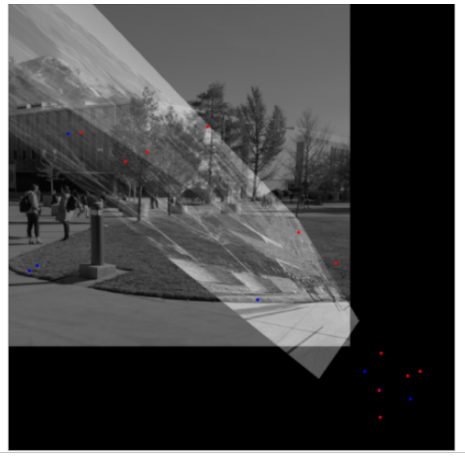


Part C: Estimating Pairwise Homographies using RANSAC

Use the RANSAC algorithm (Szeliski (http://szeliski.org/Book/), Ch 8.1.4), estimate the homography between each pair of images. You will need to decide whether you're going to manually specify the base image or determine in programmatically. Along with identifying the base image, you need to figure out the order in which you will composite the other images to the base.

You will need 4 pairs of best-match points to estimate a homography for each composited image. Below you will find a visualization of the RANSAC estimated homographies. Images 1, 2, and 3 have dots that are red, green and blue respectively (sorry the dots are a little small), representing the putative pairs. You can see where the homographies line up very well and in a few places (the middle vertically) they line up slightly less well.

For this section, you may use `cv2.findHomography()` or use the code from Lab 4 to generate the homography matrix. However, you must implement RANSAC yourself. Additionally, you may simply take the best homography of those randomly sampled, but as a possible improvement, you may implement a least squares optimization over the largest consesus set.


Part D: Creating the Mosaic

Begin with the base image and warp the remaining images (using the estimated homographies) to composite them onto the base image. You may use `cv2.warpPerspective()` for the backwards warping (Note: You may need to multiply your homography by a translation matrix keep it in the frame. You can then translate it back to put it on the correct part of the mosaic.).

For the ongoing campus example, here are the resulting warped images composited.

And, then with a very simple (but not ideal) compositing operation using the median of the mapped pixels.

Part A: Find Interest Points/Descriptors (20 points)

In [83]:
```python
import cv2
import numpy as np

def find_interest_points_and_descriptors(image_path):
    # Load the image
    image = cv2.imread(image_path)

    # Resize the images to the desired size
    image = cv2.resize(image,(500,900))

    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Initialize the SIFT detector
    sift = cv2.SIFT_create()

    # Detect keypoints and compute descriptors
    keypoints, descriptors = sift.detectAndCompute(gray_image, None)

    # Draw keypoints on the image
    image_with_keypoints = cv2.drawKeypoints(image, keypoints, None)

    return keypoints, descriptors, image_with_keypoints

# List of input image paths
image_paths = ['campus1.jpg', 'campus2.jpg', 'campus3.jpg']

for image_path in image_paths:
    keypoints, descriptors, image_with_keypoints = find_interest_points_and_descriptors(image_path)

    # Display or save the image with keypoints
    cv2.imshow('Image with Keypoints', image_with_keypoints)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

Part B: Matching Features (20 points)

In [84]:

```python
import cv2
import numpy as np

def find_and_match_features(campus1, campus2):
    # Load the two images
    img1 = cv2.imread(campus1, cv2.IMREAD_GRAYSCALE)
    img2 = cv2.imread(campus2, cv2.IMREAD_GRAYSCALE)

    # Resize the images to the desired size
    img1 = cv2.resize(img1, (400, 900))
    img2 = cv2.resize(img2, (400, 900))

    # Initialize the SIFT detector
    sift = cv2.SIFT_create()

    # Find keypoints and descriptors for both images
    keypoints1, descriptors1 = sift.detectAndCompute(img1, None)
    keypoints2, descriptors2 = sift.detectAndCompute(img2, None)

    # Initialize a Brute-Force Matcher
    bf = cv2.BFMatcher()

    # Find best matches using KNN (k-nearest neighbors)
    matches = bf.knnMatch(descriptors1, descriptors2, k=2)

    # Apply a ratio test to select good matches
    good_matches = []
    for m, n in matches:
        if m.distance < 0.75 * n.distance:
            good_matches.append(m)

    # Draw the matches
    img_matches = cv2.drawMatches(img1, keypoints1, img2, keypoints2, good_matches, None)

    return img_matches

# List of image pairs for feature matching
image_pairs = [('campus1.jpg', 'campus3.jpg'), ('campus2.jpg', 'campus3.jpg')]

# Create an empty canvas to display the matched images
output_canvas = None

for campus1, campus2 in image_pairs:
```

```
    img_matches = find_and_match_features(campus1, campus2)

    # Stack the matched images side by side
    if output_canvas is None:
        output_canvas = img_matches
    else:
        output_canvas = np.hstack((output_canvas, img_matches))

# Display or save the output canvas with matches
cv2.imshow('Feature Matches', output_canvas)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Part C: Estimating Pairwise Homographies using RANSAC (30 points)

In [85]:

```python
import cv2
import numpy as np

def estimate_homography_RANSAC(src_pts, dst_pts, ransac_threshold=5.0):
    # Use RANSAC to estimate the homography matrix
    M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, ransac_threshold)
    return M, mask

# List of image pairs for feature matching
image_pairs = [('campus1.jpg', 'campus3.jpg'), ('campus2.jpg', 'campus3.jpg')]

# Create an empty canvas to display the matched images
output_canvas = None

# List of images and their corresponding keypoints and descriptors
image_paths = ['campus1.jpg', 'campus2.jpg', 'campus3.jpg']
keypoints_list = []
descriptors_list = []

# Populate keypoints and descriptors for each image
for image_path in image_paths:
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    image = cv2.resize(image, (400, 900))
    sift = cv2.SIFT_create()
    keypoints, descriptors = sift.detectAndCompute(image, None)
    keypoints_list.append(keypoints)
    descriptors_list.append(descriptors)


# Determine the base image (e.g., the image with the most keypoints)
base_image_idx = 2 #np.argmax([len(keypoints) for keypoints in keypoints_list])

# Initialize the list to store pairwise homographies
homographies = []

# Loop through the images to estimate pairwise homographies
for idx, (keypoints1, descriptors1) in enumerate(zip(keypoints_list, descriptors_list)):
    print(idx)

    if idx != base_image_idx:
        # Match keypoints between the base image and the current image
        bf = cv2.BFMatcher()
        matches = bf.knnMatch(descriptors_list[base_image_idx], descriptors1, k=2)
```

```
        # Apply a ratio test to select good matches
        good_matches = []
        for m, n in matches:
            if m.distance < 0.75 * n.distance:
                good_matches.append(m)

        # Extract matched keypoints and their corresponding points
        src_pts = np.float32([keypoints_list[base_image_idx][m.queryIdx].pt for m in good_matches]).reshape(-1
        dst_pts = np.float32([keypoints1[m.trainIdx].pt for m in good_matches]).reshape(-1, 1, 2)

        # Estimate homography using RANSAC
        homography, mask = estimate_homography_RANSAC(src_pts, dst_pts, ransac_threshold=5.0)
        homographies.append(homography)

        # Warp the non-base images onto the base image
        image = cv2.imread(image_paths[idx])
        image = cv2.resize(image, (400, 900))
        warped_image = cv2.warpPerspective(image, homography, (image.shape[1], image.shape[0]))

        # Blend the warped image with the base image
        if output_canvas is None:
            output_canvas = warped_image
        else:
            output_canvas = cv2.addWeighted(output_canvas, 0.5, warped_image, 0.5, 0)

# Display or save the final panorama
cv2.imshow('Panorama', output_canvas)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

```
0
1
2
```

Part D: Creating the Mosaic (30 points)

In [87]:

```
import cv2
import numpy as np

# List of image paths and their corresponding homographies
image_paths = ['campus1.jpg', 'campus2.jpg', 'campus3.jpg']
homographies = []  # List of homography matrices obtained in the previous step

# Read the base image (the image that doesn't need warping)
base_image = cv2.imread(image_paths[0])
image = cv2.resize(image, (400, 900))

# Initialize the final mosaic as the base image
mosaic = base_image.copy()

# Loop through the other images and warp them onto the mosaic
for i in range(1, len(image_paths)):
    image = cv2.imread(image_paths[i])

    # Calculate the homography for the current image
    img1 = base_image  # Use the base image as the first image
    img2 = image

    # Detect keypoints and compute descriptors
    orb = cv2.ORB_create()
    keypoints1, descriptors1 = orb.detectAndCompute(img1, None)
    keypoints2, descriptors2 = orb.detectAndCompute(img2, None)

    # Create a Brute-Force Matcher
    bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

    # Match descriptors
    matches = bf.match(descriptors1, descriptors2)

    # Sort the matches by distance
    matches = sorted(matches, key=lambda x: x.distance)

    # Get the corresponding points from the matches
    points1 = np.float32([keypoints1[match.queryIdx].pt for match in matches]).reshape(-1, 1, 2)
    points2 = np.float32([keypoints2[match.trainIdx].pt for match in matches]).reshape(-1, 1, 2)

    # Compute the homography matrix using RANSAC
    homography, _ = cv2.findHomography(points1, points2, cv2.RANSAC, 5.0)
    homographies.append(homography)
```

```
    # Add a translation to keep the warped image in frame
    translation_matrix = np.zeros((3, 3), dtype=np.float64)
    translation_matrix[0, 2] = -homography[0, 2]  # Translate in the x-direction
    translation_matrix[1, 2] = -homography[1, 2]  # Translate in the y-direction
    translation_matrix[2, 2] = 1  # Set the bottom-right element to 1

    # Apply the translation
    warped_image = cv2.warpPerspective(image, translation_matrix, (image.shape[1], image.shape[0]))

    # Simple compositing by taking the median of overlapping pixels
    mosaic = cv2.addWeighted(mosaic, 0.5, warped_image, 0.5, 0)

# Display or save the final mosaic
cv2.imshow('Mosaic', mosaic)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

In [ ]: