Homework 2 : Gianni Spiga

In [1]:

```
%load_ext nb_black
```

In [2]:

```
import math
import numpy as np
import scipy as sc
import scipy.stats as ss
from scipy import ndimage
from scipy.stats import norm
from scipy.ndimage import rank_filter
from scipy.signal import convolve2d

# from skimage import filters
from skimage.color import rgb2gray
from PIL import Image
from IPython.display import display
from matplotlib import pyplot as plt
import plotly.express as px
import cv2

from numpy.random import default_rng

rng = default_rng()
```

In [3]:

```
# Functions
def dist2(x, c):
    """
    Calculates squared distance between two sets of points.

    Parameters
    ----------
    x: numpy.ndarray
        Data of shape `(ndata, dimx)`
    c: numpy.ndarray
        Centers of shape `(ncenters, dimc)`

    Returns
    -------
    n2: numpy.ndarray
        Squared distances between each pair of data from x and c, of shape
        `(ndata, ncenters)`
    """
    assert (
        x.shape[1] == c.shape[1]
    ), "Data dimension does not match dimension of centers"

    x = np.expand_dims(x, axis=0)  # new shape will be `(1, ndata, dimx)`
    c = np.expand_dims(c, axis=1)  # new shape will be `(ncenters, 1, dimc)`

    # We will now use broadcasting to easily calculate pairwise distances
    n2 = np.sum((x - c) ** 2, axis=-1)
```

```python
    return n2


def gen_dgauss(sigma):
    """
    Generates the horizontally and vertically differentiated Gaussian filter

    Parameters
    ----------
    sigma: float
        Standard deviation of the Gaussian distribution

    Returns
    -------
    Gx: numpy.ndarray
        First degree derivative of the Gaussian filter across rows
    Gy: numpy.ndarray
        First degree derivative of the Gaussian filter across columns
    """
    f_wid = 4 * np.floor(sigma)
    G = norm.pdf(np.arange(-f_wid, f_wid + 1), loc=0, scale=sigma).reshape(-1, 1)
    G = G.T * G
    Gx, Gy = np.gradient(G)

    Gx = Gx * 2 / np.abs(Gx).sum()
    Gy = Gy * 2 / np.abs(Gy).sum()

    return Gx, Gy


def find_sift(I, circles, enlarge_factor=1.5):
    """
    Compute non-rotation-invariant SIFT descriptors of a set of circles

    Parameters
    ----------
    I: numpy.ndarray
        Image
    circles: numpy.ndarray
        An array of shape `(ncircles, 3)` where ncircles is the number of
        circles, and each circle is defined by (x, y, r), where r is the radius
        of the cirlce
    enlarge_factor: float
        Factor which indicates by how much to enlarge the radius of the circle
        before computing the descriptor (a factor of 1.5 or large is usually
        necessary for best performance)

    Returns
    -------
    sift_arr: numpy.ndarray
        Array of SIFT descriptors of shape `(ncircles, 128)`
    """
    assert (
        circles.ndim == 2 and circles.shape[1] == 3
    ), "Use circles array (keypoints array) of correct shape"
    I = I.astype(np.float64)
    if I.ndim == 3:
        I = rgb2gray(I)
```

```
NUM_ANGLES = 8
NUM_BINS = 4
NUM_SAMPLES = NUM_BINS * NUM_BINS
ALPHA = 9
SIGMA_EDGE = 1

ANGLE_STEP = 2 * np.pi / NUM_ANGLES
angles = np.arange(0, 2 * np.pi, ANGLE_STEP)

height, width = I.shape[:2]
num_pts = circles.shape[0]

sift_arr = np.zeros((num_pts, NUM_SAMPLES * NUM_ANGLES))

Gx, Gy = gen_dgauss(SIGMA_EDGE)

Ix = convolve2d(I, Gx, "same")
Iy = convolve2d(I, Gy, "same")
I_mag = np.sqrt(Ix ** 2 + Iy ** 2)
I_theta = np.arctan2(Ix, Iy + 1e-12)

interval = np.arange(-1 + 1 / NUM_BINS, 1 + 1 / NUM_BINS, 2 / NUM_BINS)
gridx, gridy = np.meshgrid(interval, interval)
gridx = gridx.reshape((1, -1))
gridy = gridy.reshape((1, -1))

I_orientation = np.zeros((height, width, NUM_ANGLES))

for i in range(NUM_ANGLES):
    tmp = np.cos(I_theta - angles[i]) ** ALPHA
    tmp = tmp * (tmp > 0)

    I_orientation[:, :, i] = tmp * I_mag

for i in range(num_pts):
    cx, cy = circles[i, :2]
    r = circles[i, 2]

    gridx_t = gridx * r + cx
    gridy_t = gridy * r + cy
    grid_res = 2.0 / NUM_BINS * r

    x_lo = np.floor(np.max([cx - r - grid_res / 2, 0])).astype(np.int32)
    x_hi = np.ceil(np.min([cx + r + grid_res / 2, width])).astype(np.int32)
    y_lo = np.floor(np.max([cy - r - grid_res / 2, 0])).astype(np.int32)
    y_hi = np.ceil(np.min([cy + r + grid_res / 2, height])).astype(np.int32)

    grid_px, grid_py = np.meshgrid(
        np.arange(x_lo, x_hi, 1), np.arange(y_lo, y_hi, 1)
    )
    grid_px = grid_px.reshape((-1, 1))
    grid_py = grid_py.reshape((-1, 1))

    dist_px = np.abs(grid_px - gridx_t)
    dist_py = np.abs(grid_py - gridy_t)

    weight_x = dist_px / (grid_res + 1e-12)
    weight_x = (1 - weight_x) * (weight_x <= 1)
    weight_y = dist_py / (grid_res + 1e-12)
    weight_y = (1 - weight_y) * (weight_y <= 1)
```

```
        weights = weight_x * weight_y

        curr_sift = np.zeros((NUM_ANGLES, NUM_SAMPLES))
        for j in range(NUM_ANGLES):
            tmp = I_orientation[y_lo:y_hi, x_lo:x_hi, j].reshape((-1, 1))
            curr_sift[j, :] = (tmp * weights).sum(axis=0)
        sift_arr[i, :] = curr_sift.flatten()

    tmp = np.sqrt(np.sum(sift_arr ** 2, axis=-1))
    if np.sum(tmp > 1) > 0:
        sift_arr_norm = sift_arr[tmp > 1, :]
        sift_arr_norm /= tmp[tmp > 1].reshape(-1, 1)

        sift_arr_norm = np.clip(sift_arr_norm, sift_arr_norm.min(), 0.2)

        sift_arr_norm /= np.sqrt(np.sum(sift_arr_norm ** 2, axis=-1, keepdims=True))

        sift_arr[tmp > 1, :] = sift_arr_norm

    return sift_arr


def harris(im, sigma, thresh=None, radius=None):
    """
    Harris corner detector

    Parameters
    ----------
    im: numpy.ndarray
        Image to be processed
    sigma: float
        Standard deviation of smoothing Gaussian
    thresh: float (optional)
    radius: float (optional)
        Radius of region considered in non-maximal suppression

    Returns
    -------
    cim: numpy.ndarray
        Binary image marking corners
    r: numpy.ndarray
        Row coordinates of corner points. Returned only if none of `thresh` and
        `radius` are None.
    c: numpy.ndarray
        Column coordinates of corner points. Returned only if none of `thresh`
        and `radius` are None.
    """
    if im.ndim == 3:
        im = rgb2gray(im)

    dx = np.tile([[-1, 0, 1]], [3, 1])
    dy = dx.T

    Ix = convolve2d(im, dx, "same")
    Iy = convolve2d(im, dy, "same")

    f_wid = np.round(3 * np.floor(sigma))
    G = norm.pdf(np.arange(-f_wid, f_wid + 1), loc=0, scale=sigma).reshape(-1, 1)
    G = G.T * G
    G /= G.sum()
    Ix2 = convolve2d(Ix ** 2, G, "same")
    Iy2 = convolve2d(Iy ** 2, G, "same")
    Ixy = convolve2d(Ix * Iy, G, "same")

    cim = (Ix2 * Iy2 - Ixy ** 2) / (Ix2 + Iy2 + 1e-12)

    if thresh is None or radius is None:
        return cim
    else:
        size = int(2 * radius + 1)
        mx = rank_filter(cim, -1, size=size)
        cim = (cim == mx) & (cim > thresh)

        r, c = cim.nonzero()

        return cim, r, c


if __name__ == "__main__":
    Gx, Gy = gen_dgauss(3.2)
    print(f"Gx.shape: {Gx.shape}")
    I = np.random.random((480, 640, 3)) * 255
    circles = np.vstack(
        [
            np.random.randint(1, 480, 25),
            np.random.randint(1, 640, 25),
            15 * np.random.random(25),
        ]
    ).T

    sift_arr = find_sift(I, circles)
    print(sift_arr.shape)

    cim, r, c = harris(I, 3.2, thresh=5, radius=3)

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

```
Gx.shape: (25, 25)
(25, 128)
cim.shape: (480, 640)
```

1.)

In [4]:

```
utowerLeftColor = Image.open("hw2_data/uttower_left.jpg")
utowerRightColor = Image.open("hw2_data/uttower_right.jpg")

utowerLeft = Image.open("hw2_data/uttower_left.jpg").convert("L")
utowerRight = Image.open("hw2_data/uttower_right.jpg").convert("L")

# Convert to arrays of double
utowerLeft = np.asarray(
    utowerLeft, dtype=np.float32
)  # np.double does not allow the harris corner detection to work
utowerRight = np.asarray(utowerRight, dtype=np.float32)

display(px.imshow(utowerLeft, color_continuous_scale="gray"))
display(px.imshow(utowerRight, color_continuous_scale="gray"))
```

0
100
200
300

0
100
200
300

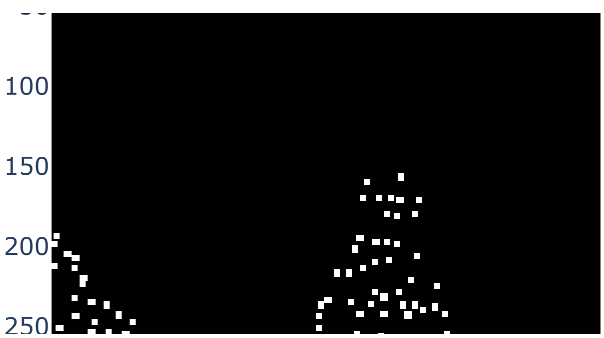
2.)

In [5]:

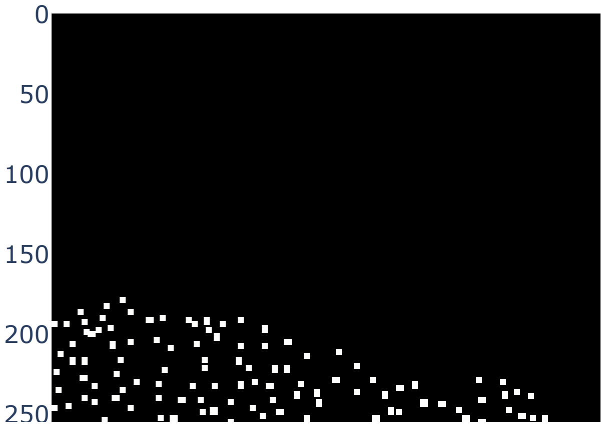
```
### OpenCV which is wrong

# # Harris Detection for the left
# leftHCD = cv.cornerHarris(utowerLeft, 2, 3, 0.04)
# leftHCD = cv.dilate(leftHCD, None)
# utowerLeft[leftHCD > 0.01 * leftHCD.max()] = 255

# # Harris Detection for the right
# rightHCD = cv.cornerHarris(utowerRight, 2, 3, 0.04)
# rightHCD = cv.dilate(rightHCD, None)
# utowerRight[rightHCD > 0.01 * rightHCD.max()] = 255

# display(px.imshow(utowerLeft, color_continuous_scale="gray"))
# display(px.imshow(utowerRight, color_continuous_scale="gray"))
```

In [126…

```
# Harris Detection for the left
cimL, rL, cL = harris(utowerLeft, 2, thresh=50, radius=2)
# From Piazza @169
vis_cimL = np.zeros_like(cim).astype(np.uint8)
for i, j in zip(rL, cL):
    vis_cimL[i - 2 : i + 2, j - 2 : j + 2] = 255
fig = px.imshow(vis_cimL, color_continuous_scale="gray")
fig.write_image("LeftHarris.png")
display(fig)

# How many points
print(rL.shape)

# Harris Detection for the right
cimR, rR, cR = harris(utowerRight, 2, thresh=50, radius=2)
vis_cimR = np.zeros_like(cim).astype(np.uint8)
for i, j in zip(rR, cR):
    vis_cimR[i - 2 : i + 2, j - 2 : j + 2] = 255
fig = px.imshow(vis_cimR, color_continuous_scale="gray")
fig.write_image("RightHarris.png")
display(fig)

# How many points
print(rR.shape)
```



```
(2715,)
```



```
(2882,)
```

3.)

In [7]:

```
def standardizeRows(matrix):
    mat = matrix
    for i in range(len(mat)):
        if np.std(mat[i,]) != 0:
            matrix[i,] = (mat[i,] - np.mean(mat[i,])) / np.std(mat[i,])
        else:
            mat[i,] = np.zeros(len(mat[i,]))
    return mat


# Look up cv2.rectangle


def featureDescription(image, rad, row, col):
    featCount = len(row)  # feature quantity
    # desc = np.zeros(shape=(featCount, (2 * rad - 1) ** 2))
    desc = np.zeros(shape=(featCount, (rad) ** 2))

    # This is for keeping track of all indices, we only need the top left corner and bo
    indexList = np.array([]).reshape(-1, 4)

    # Zero padding by radius length
    padImg = np.pad(image, 2 * rad + 1)

    for i in range(featCount):
        # row and column neighbourhood
        #         rowNH = list(range(row[i], row[i] + 2 * rad))
        #         colNH = list(range(col[i], col[i] + 2 * rad))

        rowNH = list(range(row[i], row[i] + rad))
        colNH = list(range(col[i], col[i] + rad))

        ixgrid = np.ix_(rowNH, colNH)
        # Create neighbourhood matrix
        ixgrid = np.ix_(rowNH, colNH)
        #         print("last entry of row nh is ", rowNH[-1])
        submat = padImg[ixgrid]
        # submat = padImg[rowNH[0] : rowNH[-1], colNH[0] : colNH[-1]]
        # print("submat is ", submat)

        # Save location
        newrow = [int(rowNH[0]), int(colNH[0]), int(rowNH[-1]), int(colNH[-1])]
        indexList = np.vstack([indexList, newrow])
        # print("IndexList is \n", indexList)

        # Flatten Matrix by Column (fortran flattening)
        nbhd = submat.flatten(order="F")
        # print(nbhd)
        desc[i,] = nbhd

    # Standardize descriptors
    standardizeRows(desc)
    # print("index list is ", indexList.reshape(-1, 5))
    return desc, indexList
```

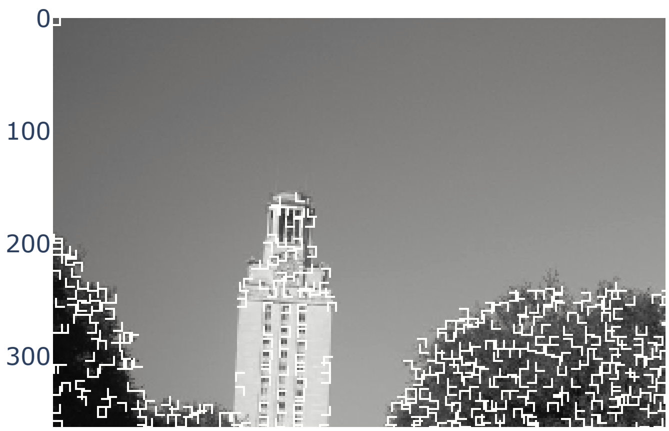
```
leftDesc3, leftIDX3 = featureDescription(utowerLeft, 3, rL, cL)
rightDesc3, rightIDX3 = featureDescription(utowerRight, 3, rR, cR)

leftDesc4, leftIDX4 = featureDescription(utowerLeft, 4, rL, cL)
rightDesc4, rightIDX4 = featureDescription(utowerRight, 4, rR, cR)

leftDesc5, leftIDX5 = featureDescription(utowerLeft, 5, rL, cL)
rightDesc5, rightIDX5 = featureDescription(utowerRight, 5, rR, cR)

print("LEFT PICTURE\n")
print("Neighbourhood Size = 3:\n", leftDesc3)
print("Neighbourhood Size = 4:\n", leftDesc4)
print("Neighbourhood Size = 5:\n", leftDesc5)

print("\nRIGHT PICTURE\n")
print("Neighbourhood Size = 3:\n", rightDesc3)
print("Neighbourhood Size = 4:\n", rightDesc4)
print("Neighbourhood Size = 5:\n", rightDesc5)
```

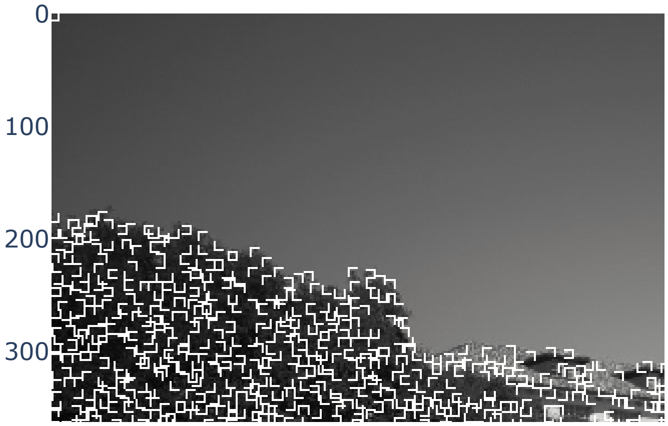
```
LEFT PICTURE

Neighbourhood Size = 3:
 [[ 0.          0.          0.         ...  0.          0.
   0.        ]
 [ 0.          0.          0.         ...  0.          0.
   0.        ]
 [ 0.          0.          2.12132034 ...  0.          0.
   0.        ]
 ...
 [ 0.87605923  0.60133683 -1.29424779 ...  0.87605923  0.46397562
  -1.48655347]
 [ 1.22838454  0.09449112 -1.32287566 ...  1.22838454  0.37796447
  -0.75592895]
 [-0.5         0.1        -0.8        ...  1.          1.
  -0.5       ]]
Neighbourhood Size = 4:
 [[ 0.          0.          0.         ...  0.          0.
   0.        ]
 [ 0.          0.          0.         ...  0.          0.
   0.        ]
 [-2.32379001 -0.25819889 -0.25819889 ...  1.80739223 -0.25819889
  -0.25819889]
 ...
 [ 0.46235805  0.17225104  0.31730454 ... -0.40796298  0.02719753
  -2.29365855]
 [-0.17259885  1.36161317  2.28214039 ... -0.47944126 -0.47944126
  -1.70681088]
 [-0.55167728  0.70929937  1.02454353 ...  2.28552018  1.02454353
   0.07881104]]
Neighbourhood Size = 5:
 [[ 0.          0.          0.         ...  0.          0.
   0.        ]
 [ 0.          0.          0.         ...  0.          0.
   0.        ]
 [-0.8660254   0.57735027  0.57735027 ...  0.57735027  0.57735027
   0.57735027]
 ...
 [ 1.95220673  1.35459242 -0.03984095 ...  0.95618289 -0.63745526
```

```
  -1.43427433]
 [-2.31744652 -2.06555016 -1.05796472 ... -0.30227563 -0.30227563
  -0.05037927]
 [ 0.20834029 -1.59460452  0.20834029 ... -0.99362292 -0.39264131
  -0.99362292]]

RIGHT PICTURE

Neighbourhood Size = 3:
 [[ 0.          0.          0.         ...  0.          0.
   0.        ]
 [ 0.          0.          0.         ...  0.          0.
   0.        ]
 [ 0.35355339  0.35355339  0.35355339 ... -2.82842712  0.35355339
   0.35355339]
 ...
 [ 1.48059236  0.94037623  0.04001601 ...  0.58023214 -0.86034421
  -1.94077647]
 [ 0.30411437  1.33050036  1.33050036 ... -1.40652895 -1.06440029
  -0.38014296]
 [ 1.36693574  1.70398839 -0.48685383 ... -1.49801177  0.01872515
  -0.48685383]]
Neighbourhood Size = 4:
 [[ 0.          0.          0.         ...  0.          0.
   0.        ]
 [ 0.          0.          0.         ...  0.          0.
   0.        ]
 [ 0.37796447  0.37796447  0.37796447 ... -2.64575131  0.37796447
   0.37796447]
 ...
 [ 2.16351833  1.39851737  0.63351642 ... -1.08773573 -1.27898597
  -1.85273669]
 [-0.64483142  0.05862104 -0.17586311 ... -1.58276803 -0.17586311
   0.52758934]
 [ 1.08836485  0.33668841  0.83780603 ... -1.04138507  0.211409
   0.08612959]]
Neighbourhood Size = 5:
 [[ 0.          0.          0.         ...  0.          0.
   0.        ]
 [ 0.          0.          0.         ...  0.          0.
   0.        ]
 [-1.040833   -1.040833   -1.040833   ...  0.96076892  0.96076892
   0.96076892]
 ...
 [-0.03204207  0.50199238  0.23497516 ... -1.90116263 -2.16817985
  -1.90116263]
 [ 1.12402767 -0.26709568  1.12402767 ... -0.82354502 -1.10176969
   1.12402767]
 [ 0.04502003  1.17052088  0.84894921 ... -2.68833916 -1.56283832
   0.68816337]]
```

In [124…

```
utower_left_copy = utowerLeft.copy()
for i in range(len(leftIDX3)):
    # print("\nNEW ITERATION \n")
    start_points = (
        int(leftIDX3[i][1]) - 3,
        int(leftIDX3[i][0]) - 3,
    )
    # print("start points is", start_points)
    end_points = (
        int(leftIDX3[i][3]) + 3,
        int(leftIDX3[i][2]) + 3,
    )
    # print("end points is ", end_points)
    image_point = cv2.rectangle(
        utower_left_copy, start_points, end_points, (255, 0, 0), 1
    )
fig = px.imshow(image_point, color_continuous_scale="gray")
fig.write_image("LeftFeatures.png")
display(fig)


utower_right_copy = utowerRight.copy()
for i in range(len(rightIDX3)):
    # print("\nNEW ITERATION \n")
    start_points = (
        int(rightIDX3[i][1]) - 3,
        int(rightIDX3[i][0]) - 3,
    )
    # print("start points is", start_points)
    end_points = (
        int(rightIDX3[i][3]) + 3,
        int(rightIDX3[i][2]) + 3,
    )
    # print("end points is ", end_points)
    image_point2 = cv2.rectangle(
        utower_right_copy, start_points, end_points, (255, 0, 0), 1
    )
fig = px.imshow(image_point2, color_continuous_scale="gray")
fig.write_image("RightFeatures.png")
display(fig)
```





4.)

In [9]:

```
print("Euclidean Distance of Descriptors:\n")
print(dist2(leftDesc5, rightDesc5))

# print("\nEuclidean Distance of Standardized Descriptors:\n")
# print(dist2(standardizeRows(leftDesc), standardizeRows(rightDesc)))
```

```
Euclidean Distance of Descriptors:

[[ 0.          0.         25.         ... 25.         25.
  25.        ]
 [ 0.          0.         25.         ... 25.         25.
  25.        ]
 [25.         25.         25.73186642 ... 86.36405261 19.34486859
  59.20639314]
 ...
 [25.         25.         62.1788348  ... 54.85102869 44.29629545
  29.33970837]
 [25.         25.         31.04529559 ... 60.95171935 42.5430856
  35.40163488]
 [25.         25.         62.90333457 ... 63.04233299 44.3784103
  36.42551187]]
```

5.)

In [65]:

```python
def findMatches(leftDesc, rightDesc, threshold=0, n_shortest=0):
    # Calculate distances between pairs and features
    print("leftDesc shape is ", leftDesc.shape)
    print("rightDesc shape is ", rightDesc.shape)

    dist = dist2(leftDesc, rightDesc)
    print("dist2 shape is", dist)
    # Remove all values below a certain threshold
    if threshold > 0:
        # Returns row and column indexes as separate lists where below threshold
        ind = np.argwhere(dist < threshold)
        print(ind)
        c, r = ind[:, 0], ind[:, 1]
        matches = 1

    if n_shortest > 0:
        # Recieve n shortest distances from sorted list
        # argsort will return indices
        sort = np.argsort(dist.flatten(order="F"))
        matches = sort[:n_shortest]
        c, r = np.unravel_index(matches, dist.shape)
    return r, c, matches


leftIndex, rightIndex, matches = findMatches(leftDesc4, rightDesc4, threshold=2)
# leftIndex, rightIndex, matches = findMatches(leftDesc4, rightDesc4, n_shortest=7000)
print("leftIndex shape is ", leftIndex.shape)
print("rightIndex shape is ", rightIndex.shape)
```

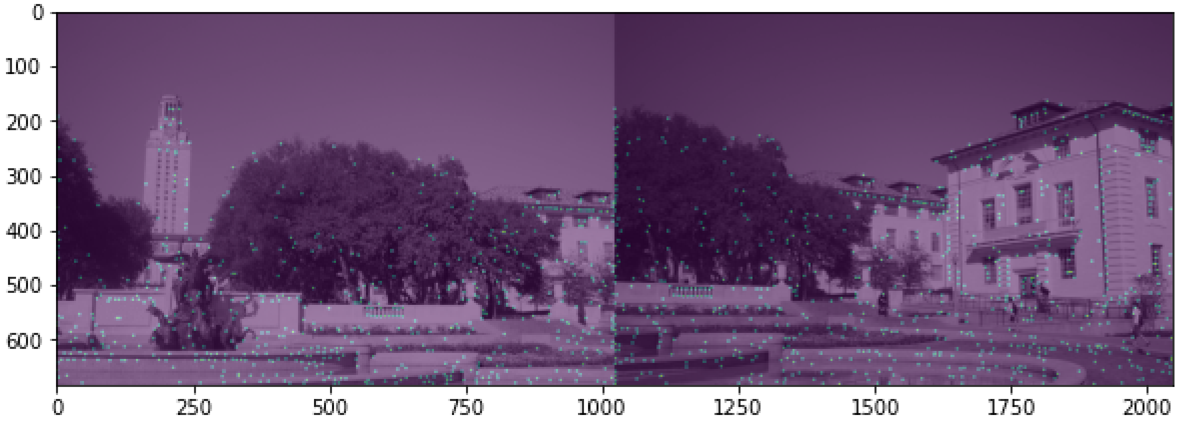
```
leftDesc shape is  (2715, 16)
rightDesc shape is  (2882, 16)
dist2 shape is [[ 0.          0.         16.         ... 16.         16.
  16.        ]
 [ 0.          0.         16.         ... 16.         16.
  16.        ]
 [16.         16.         53.86016163 ... 37.59215732 28.05683204
  52.01738375]
 ...
 [16.         16.         44.93771702 ... 17.95932444 12.30428879
  42.52069597]
 [16.         16.         38.29652898 ... 52.11854063 31.53232707
  45.45339638]
 [16.         16.         47.33247979 ... 50.17448866 25.60437287
  38.77315269]]
[[   0    0]
 [   0    1]
 [   0   12]
 ...
 [2874 2645]
 [2874 2658]
 [2880 1083]]
leftIndex shape is  (7271,)
rightIndex shape is  (7271,)
```

In [125…

```
# Indexing issues out of range, this is quick solution that removes them
# leftIndex = leftIndex[leftIndex < len(rL)]
# rightIndex = rightIndex[rightIndex < len(rR)]

# Matches for left image
rowMatchLeft = rL[leftIndex]
colMatchLeft = cL[leftIndex]

# Matches for right image
rowMatchRight = rR[rightIndex]
colMatchRight = cR[rightIndex]

print("Row match left \n", rowMatchLeft.shape)
# Plot image matching
colWidth = utowerLeft.shape[1]
plotLeft = np.concatenate(
    (rowMatchLeft[np.newaxis].T, colMatchLeft[np.newaxis].T), axis=1
)
plotRight = np.concatenate(
    (rowMatchRight[np.newaxis].T, colMatchRight[np.newaxis].T + colWidth), axis=1
)
print("plotRight is ", plotRight)
joinedImages = np.concatenate((utowerLeft, utowerRight), axis=1)
# display(px.imshow(joinedImages, color_continuous_scale="gray"))

markingMatches = np.zeros_like(joinedImages).astype(np.uint8)
for i, j in zip(rowMatchLeft, colMatchLeft):
    markingMatches[i - 2 : i + 2, j - 2 : j + 2] = 255

for i, j in zip(rowMatchRight, colMatchRight + colWidth):
    markingMatches[i - 2 : i + 2, j - 2 : j + 2] = 255

plt.figure(figsize=(10, 10))
display(plt.imshow(joinedImages, cmap="gray"))
display(plt.imshow(markingMatches, alpha=0.5))
plt.savefig("BestMatches.png")

plotLeft = np.concatenate((plotLeft, np.ones((len(rowMatchLeft), 1))), axis=1)
plotRight = np.concatenate((plotRight, np.ones((len(rowMatchRight), 1))), axis=1)

print("PlotLeft is \n", plotLeft.shape)
```

```
Row match left
 (7271,)
plotRight is  [[   1 1025]
 [   1 1025]
 [   1 1025]
 ...
 [ 681 1437]
 [ 681 1437]
 [ 681 1904]]
<matplotlib.image.AxesImage at 0x21e98132340>
<matplotlib.image.AxesImage at 0x21e98bd9670>
PlotLeft is
```

```
(7271, 3)
```



6.)

In [121…

```
def ransac(leftImgInf, rightImgInf):
    matches = np.shape(leftImgInf)[0]
    #print(matches)
    #We will do 200 iterations
    inliers = np.zeros(shape = (200, 1))

    #keep track of models
    masterList = np.array([])
    #Keep track of number of inliers
    inlierCount = np.array([])

    for i in range(matches):
        #randomly sample indices
        sample = rng.choice(matches, size = 4) # "use four matches to initialize"
        #print(sample)

        leftSamp = leftImgInf[sample, ]
        rightSamp = rightImgInf[sample, ]

        X = homography(leftImgInf, rightImgInf, sample)
        residuals = findResiduals(X, leftSamp, rightSamp)
        # Filter residuals by a threshold, we will use 10 here
        inlierLoc = np.argwhere(residuals < 1000000)
        inlierCount = np.append(inlierCount, len(inlierLoc))
        #print(inlierCount)
        #inlier proportion
        inlierProp = inlierCount[i]/matches
        #print(inlierProp)
        masterList = np.append(masterList, X)
        # Look for ambiguous points
#         if inlierProp >= 0.3: #Look for iterations where at least a quarter of the in
#             rowInliers = leftImgInf[inlierLoc, :]
#             colInliers = rightImgInf[inlierLoc, :]
#             masterList = np.append(masterList, homography(rowInliers.flatten().reshap
#             print(homography(rowInliers.reshape((-1, 3)), colInliers.reshape((-1, 3))

    #Find where there was most matches and get the first entry(if there was more than o
    mostMatchesLoc = np.argwhere(inlierCount == max(inlierCount))
    print(masterList)
    topMatches = masterList[mostMatchesLoc]


    #calculate residuals again
    #residualRepeat = findResiduals(np.transpose(topMatches.reshape(-1, 3)), leftImgInf
#     residualRepeat = findResiduals(topMatches, leftImgInf, rightImgInf)
#     bestInlierLoc = np.argwhere(residualRepeat < 1000000)
    return topMatches, mostMatchesLoc

def homography(leftSamp, rightSamp, sample):
    ### Homography
    #print("leftSamp is ", leftSamp)
    A = []
    for j in range(len(sample)):
        homog1 = leftSamp[j, ]
        homog2 = rightSamp[j, ]
        zz = np.array(np.vstack((np.concatenate((np.array([0,0,0]), -1 * homog1, homog2
                                         np.concatenate((homog1, np.array([0,0,0]), -1
        for k in range(len(zz)): #append each row
                A.append(zz[k])

    #From tips and details/piazza
    A = np.array(A)
    ATA = np.transpose(A) @ A
    #reshape into 3x3
    X = np.linalg.svd(ATA)[2][8].reshape(3,3)
    X = X / X[2,2]
    return X

def findResiduals(X, leftImgInf, rightImgInf):
    ### Find residuals
    #Transform points
#     trans = leftImgInf @ X
#     #We want the third column
#     scaleCol = trans[:,2]
#     rightHomo = rightImgInf[:,2]
#     Xdist = trans[:,0] / (scaleCol - rightImgInf[:,0]) / rightHomo
#     Ydist = trans[:,1] / (scaleCol - rightImgInf[:,1]) / rightHomo
#     residuals = Xdist * Xdist + Ydist * Ydist

    residuals = np.mean(dist2(leftImgInf, rightImgInf))
    #print(residuals)
    return residuals

#print(plotRight)
ransac(plotLeft, plotRight)
```

```
[-0.23850473  0.07645959  0.99999992 ... -0.23850473  0.07645959
  1.        ]
```

Out[121…

```
(array([[ 7.64595872e-02],
        [-2.44467344e+02],
        [ 7.64595872e-02],
        ...,
        [ 7.83710769e+01],
        [ 7.64595873e-02],
        [-2.38504726e-01]]),
 array([[   7],
        [  12],
        [  16],
        ...,
        [7258],
        [7264],
        [7269]], dtype=int64))
```

```
In [ ]:
```