Computer Vision (CSE 6239) assignment 02

Submited To

Dr. Sk. Mohammad Masudul Ahsan
(Professor)
(Department of Computer Science and Engineering)
(Khulna University of Engineering and Technology (KUET) Khulna 9203 Bangladesh)

Submited By

Md. Shahidul Islam
(Student ID: 1907509)

Computer Science and Engineering Discipline
Khulna University of Engineering and Technology
Khulna-9208, Bangladesh

Chapter 1

Answer to the question No 1

All program can be run online in colab:

https://colab.research.google.com/drive/12ZYc1OvALNBnE8OlXQIfTgfxKPxk31jx

1.1 Harris Corner Detection

The Harris (or Harris and Stephens) corner detection algorithm is one of the simplest corner indicators available. The general idea is to locate points where the surrounding neighborhood shows edges in more than one direction, these are then corners or interest points. The algorithm is explained here. In short, a matrix W is created from the outer product of the image gradient, this matrix is averaged over a region and then a corner response function is defined as the ratio of the determinant to the trace of W.

The idea is to consider a small window around each pixel p in an image. We want to identify all such pixel windows that are unique. Uniqueness can be measured by shifting each window by a small amount in a given direction and measuring the amount of change that occurs in the pixel values.More formally, we take the sum squared difference (SSD) of the pixel values before and after the shift and identifying pixel windows where the SSD is large for shifts in all 8 directions. Let us define the change function E(u,v) as the sum of all

the sum squared differences (SSD), where u,v are the x,y coordinates of every pixel in our 3 x 3 window and I is the intensity value of the pixel. The features in the image are all pixels that have large values of E(u,v), as defined by some threshold.

$$E(u,v) = \sum_{x,y} w(x,y)[I(x+u, y+v) - I(x,y)]^2 \tag{1.1}$$

- E is the difference between the original and the moved window.
- u is the window's displacement in the x direction
- v is the window's displacement in the y direction
- w(x, y) is the window at position (x, y). This acts like a mask. Ensuring that only the desired window is used.
- I is the intensity of the image at a position (x, y)
- I(x+u, y+v) is the intensity of the moved window
- I(x, y) is the intensity of the original

We've looking for windows that produce a large E value. To do that, we need to high values of the terms inside the square brackets.So, we maximize this term:

$$\sum_{x,y} [I(x+u, y+v) - I(x,y)]^2 \tag{1.2}$$

Then, we expand this term using the Taylor series. It's just a way of rewriting this term in using its derivatives.

$$E(u,v) \approx \sum_{x,y} [I(x,y) + uI_x + vI_y - I(x,y)]^2 \tag{1.3}$$

the I(x+u, y+v) changed into a totally different form $(I(x,y) + uI_x + vI_y)$ Thats the Taylor series in action. And because the Taylor series is infinite, we've ignored all terms after the first three. It gives a pretty good approximation. But it isn't the actual value.

Next, we expand the square. The I(x,y) cancels out, so its just two terms we need to square. It looks like this:

$$E(u,v) \approx \sum_{x,y} u^2 I_x^2 + 2uvI_xI_y + v^2 I_y^2 \tag{1.4}$$

Now this messy equation can be tucked up into a neat little matrix form like this:

$$E(u,v) \approx \begin{bmatrix} u & v \end{bmatrix} \left(\sum \begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix} \right) \begin{bmatrix} u \\ v \end{bmatrix} \tag{1.5}$$

the entire equation gets converted into a neat little matrix! Now, we rename the summed-matrix, and put it to be M:

$$M = \sum w(x,y) \begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix} \tag{1.6}$$

So the equation now becomes:

$$E(u,v) \approx \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix} \tag{1.7}$$

It was figured out that eigenvalues of the matrix can help determine the suitability of a

window. A score, R, is calculated for each window:

$$\begin{aligned} R &= \det M - k(\operatorname{trace} M)^2 \\ \det M &= \lambda_1 \lambda_2 \\ \operatorname{trace} M &= \lambda_1 + \lambda_2 \end{aligned} \tag{1.8}$$

$\lambda 1$ and $\lambda 2$ are the eigenvalues of M. So the values of these eigenvalues decide whether a region is a corner, edge or flat.

- When $|R|$ is small, which happens when $\lambda 1$ and $\lambda 2$ are small, the region is flat.
- When $R < 0$, which happens when $\lambda 1 >> \lambda 2$ or vice versa, the region is an edge.
- When R is large, which happens when $\lambda 1$ and $\lambda 2$ are large and $\lambda 1 \sim \lambda 2$, the region is a corner.

```
#========================== haris.py ======================
import numpy as np
import cv2
import sys
import getopt
import matplotlib.pyplot as plt


def readImage(filename):
"""
Read in an image file, errors out if we can't find the file
```

```
:param filename: The image filename.
:return: The img object in matrix form.
"""
img = cv2.imread(filename, 0)
if img is None:
print('Invalid image:' + filename)
return None
else:
print('Image successfully read...')
return img


def integralImage(img):
"""

:param img:
:return:
"""
height = img.shape[0]
width = img.shape[1]
int_image = np.zeros((height, width), np.uint64)
for y in range(height):
for x in range(width):
up = 0 if (y-1 < 0) else int_image.item((y-1, x))
left = 0 if (x-1 < 0) else int_image.item((y, x-1))
diagonal = 0 if (
```

```
x-1 < 0 or y-1 < 0) else int_image.item((y-1, x-1))
val = img.item((y, x)) + int(up) + int(left) - int(diagonal)
int_image.itemset((y, x), val)
return int_image


def adjustEdges(height, width, point):
"""
This handles the edge cases if the box's bounds are outside the image range b
:param height: Height of the image.
:param width: Width of the image.
:param point: The current point.
:return:
"""
newPoint = [point[0], point[1]]
if point[0] >= height:
newPoint[0] = height - 1

if point[1] >= width:
newPoint[1] = width - 1
return tuple(newPoint)


def findArea(int_img, a, b, c, d):
"""
Finds the area for a particular square using the integral image. See summed a
```

```
:param int_img: The
:param a: Top left corner.
:param b: Top right corner.
:param c: Bottom left corner.
:param d: Bottom right corner.
:return: The integral image.
"""
height = int_img.shape[0]
width = int_img.shape[1]
a = adjustEdges(height, width, a)
b = adjustEdges(height, width, b)
c = adjustEdges(height, width, c)
d = adjustEdges(height, width, d)

a = 0 if (a[0] < 0 or a[0] >= height) or (
a[1] < 0 or a[1] >= width) else int_img.item(a[0], a[1])
b = 0 if (b[0] < 0 or b[0] >= height) or (
b[1] < 0 or b[1] >= width) else int_img.item(b[0], b[1])
c = 0 if (c[0] < 0 or c[0] >= height) or (
c[1] < 0 or c[1] >= width) else int_img.item(c[0], c[1])
d = 0 if (d[0] < 0 or d[0] >= height) or (
d[1] < 0 or d[1] >= width) else int_img.item(d[0], d[1])

return a + d - b - c
```

```
def boxFilter(img, filterSize):
"""
Runs the subsequent box filtering steps. Prints original image, finds integra
:param img: An image in matrix form.
:param filterSize: The filter size of the matrix
:return: A final image written as finalimage.png
"""
print("Printing original image...")
print(img)
height = img.shape[0]
width = img.shape[1]
intImg = integralImage(img)
finalImg = np.ones((height, width), np.uint64)
print("Printing integral image...")
print(intImg)
cv2.imwrite("integral_image.png", intImg)
loc = filterSize//2
for y in range(height):
for x in range(width):
finalImg.itemset((y, x), findArea(intImg, (y-loc-1, x-loc-1),
(y-loc-1, x+loc), (y+loc, x-loc-1), (y+loc, x+loc))//(filterSize**2))
print("Printing final image...")
print(finalImg)
plt.imshow(finalImg,cmap='gray')

cv2.imwrite("finalimage.png", finalImg)
```

```
def main():
img = readImage("House1.jpg")
boxFilter(img, 3)



if __name__ == "__main__":
main()
#========================== END of average.py ==========
```

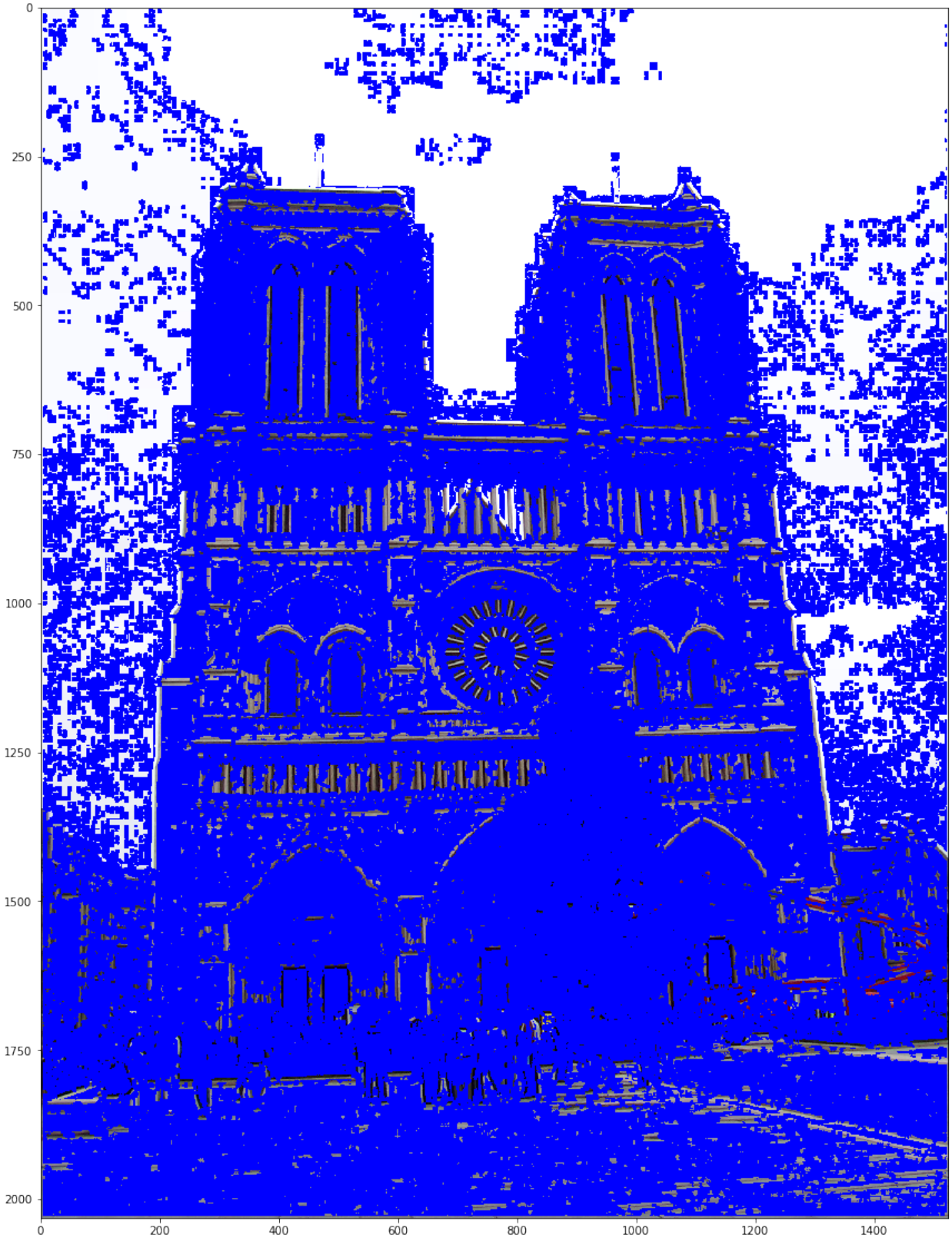


Figure 1.1: Haris corner detected image before Non-maximum suppression

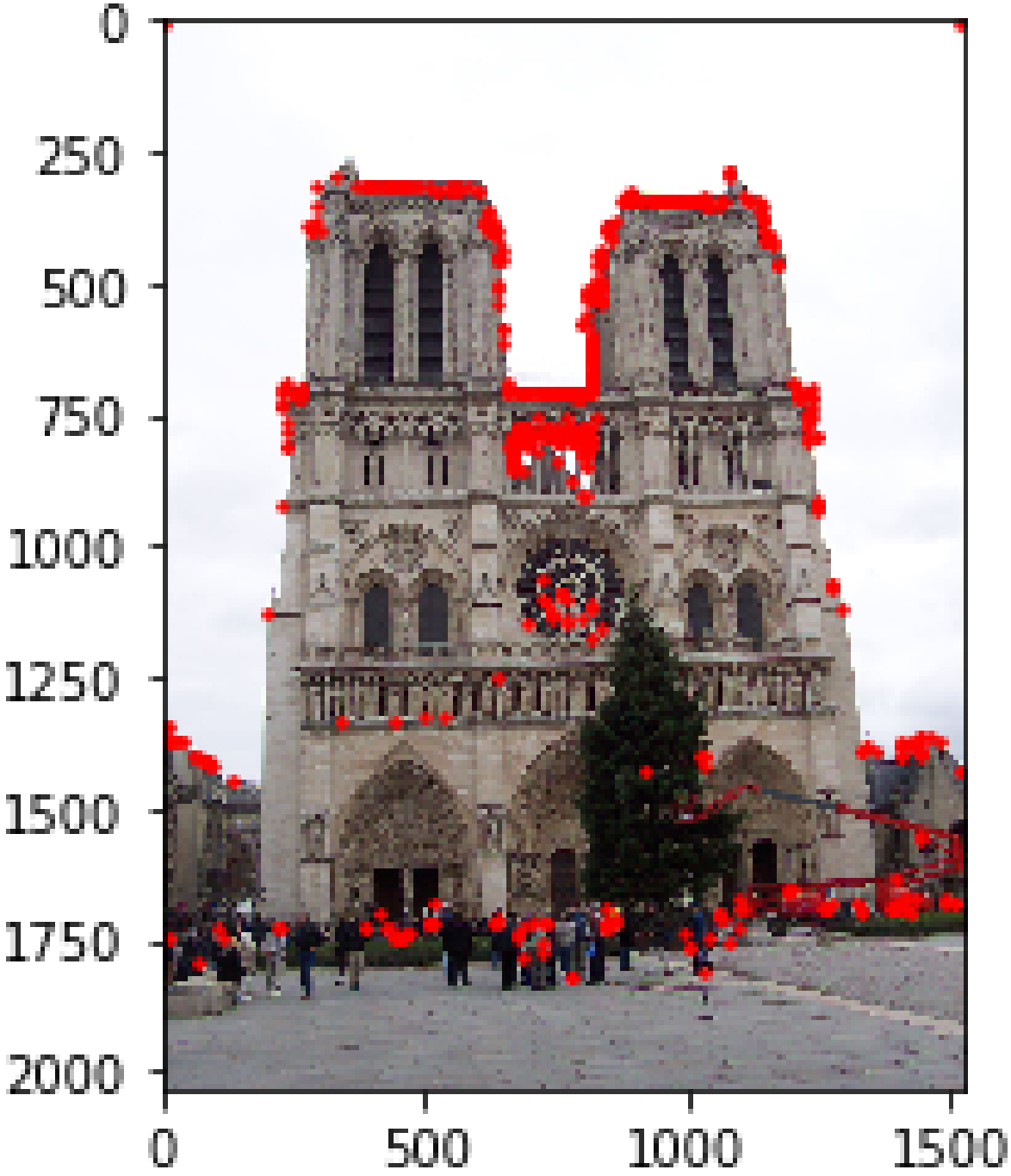


Figure 1.2: Final Haris corner detected image

Chapter 2

Answer to the question No 2

2.1 Multi-Scale Oriented Patches (MOPS) Feature Descriptor

Multi-Scale Oriented Patches (MOPS) are a minimalist design for local invariant features. They consist of a simple bias-gain normalized patch, sampled at a coarse scale relative to the interest point detection. The low frequency sampling helps to give insensitivity to noise in the interest point position. The interest points I use are multi-scale Harris corners. For each input image , a Gaussian image pyramid is formed, using a subsampling rate s = 2 and pyramid smoothing width mp= 1.0. Interest points are extracted from each level of the pyramid. First, the image is transferred to gray level, and then Gaussian pyramid is built with different octaves. The blurred images are used for the next (higher) octave to create the blurred images, and so forth. The idea is shown in the Figure 1.

The Harris matrix at level l and position (x, y) is the smoothed outer product of the gradients

$$\mathbf{H}_l(x,y) = \nabla_{\sigma_d} P_l(x,y) \nabla_{\sigma_d} P_l(x,y)^T * g_{\sigma_i}(x,y) \tag{2.1}$$

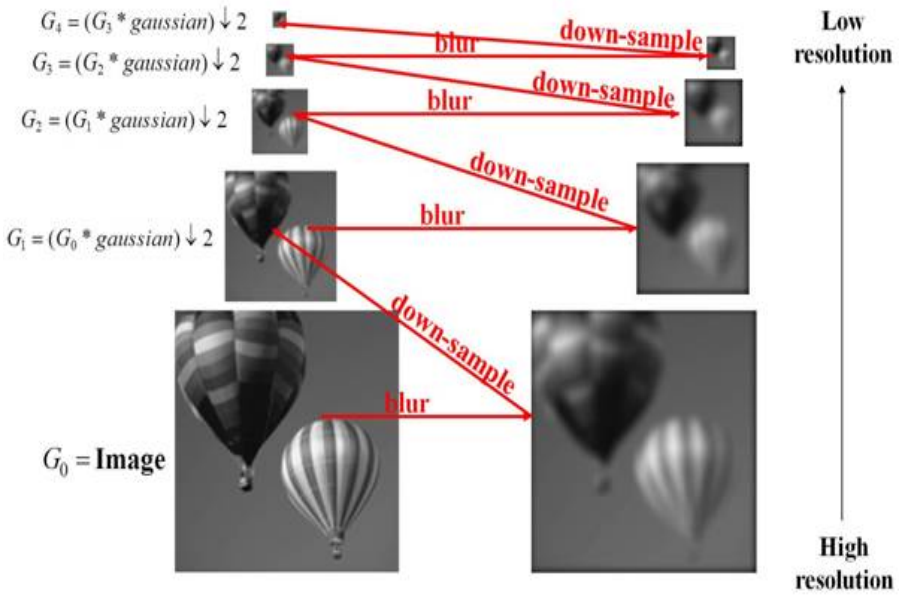


Figure 2.1: Gaussian image pyramid

Agorithm author set the integration scale $\sigma_i = 1.5$ and the derivative scale $\sigma_d = 1.0$. To find interest points, we first compute the "corner strength" function

$$f_{HM}(x, y) = \frac{\det \mathbf{H}_l(x, y)}{\operatorname{tr} \mathbf{H}_l(x, y)} = \frac{\lambda_1 \lambda_2}{\lambda_1 + \lambda_2} \tag{2.2}$$

which is the harmonic mean of the eigenvalues (λ_1, λ_2) of H. Interest points are located where the corner strength $f_{HM}(x, y)$ is a local maximum in a 3 x 3 neighbourhood, and above a threshold t = 10.0. Once local-maxima have been detected, their position is refined to sub-pixel accuracy by fitting a 2D quadratic to the corner strength function in the local 3 x 3 neighbourhood and finding its maximum. For each interest point, we also compute an orientation θ, where the orientation vector $[\cos\theta, \sin\theta] = \mathbf{u}/|\mathbf{u}|$ comes from the smoothed local gradient

```
#=================== mops.py ====================
#!/usr/bin/env python3
import cv2
```

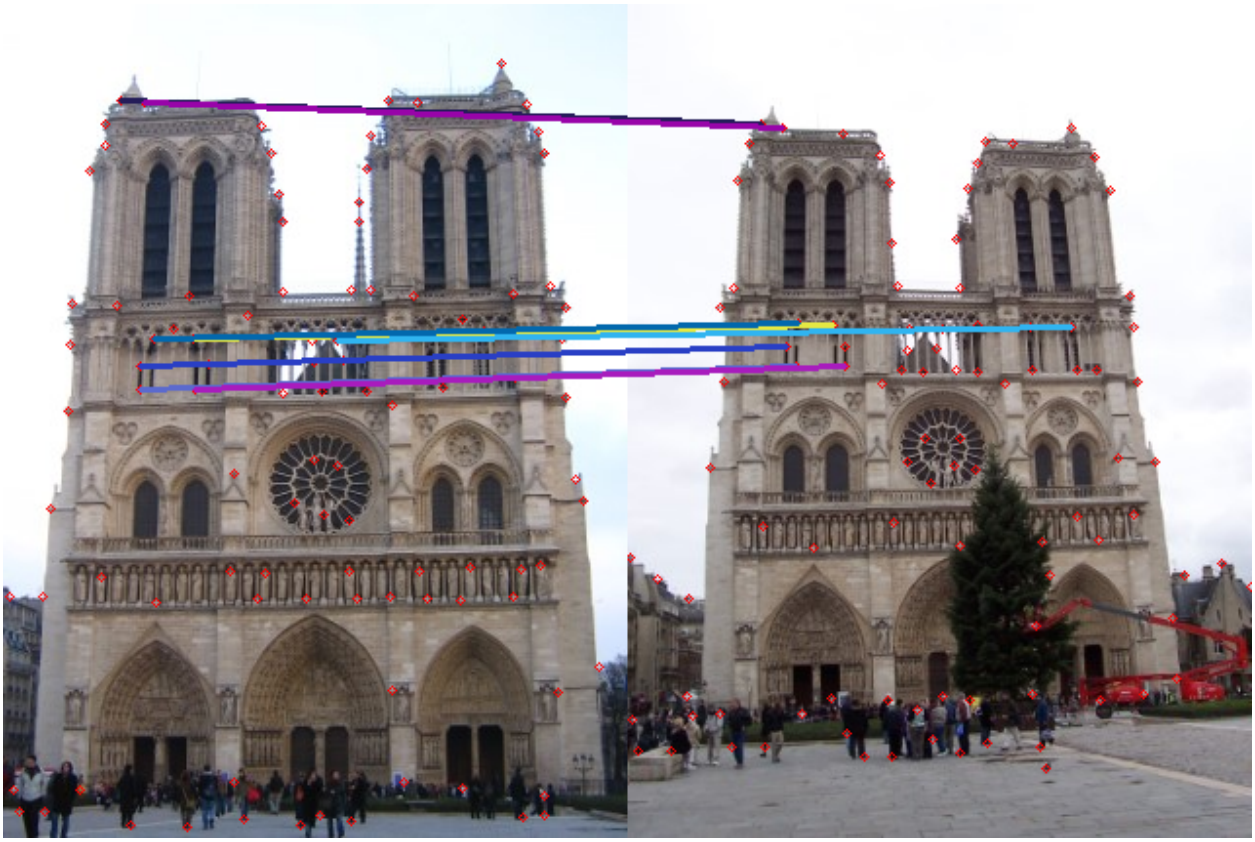


Figure 2.2: Mops detected image [using low resolution img , High resolution may detect more point]

```
import math
import pywt
import numpy as np
import argparse as ap
import matplotlib.pyplot as plt
from skimage.transform import warp, AffineTransform
from operator import itemgetter
from harris import harris, draw_corners
from utils import read_image, draw_matches, Timer
```

```
def ransac(pts1, pts2, img_l, img_r, max_iters=1000, epsilon=1):
best_matches = []
# Number of samples
N = 4

for i in range(max_iters):
# Get 4 random samples from features
idx = np.random.randint(0, len(pts1) - 1, N)
src = pts1[idx]
dst = pts2[idx]

# Calculate the homography matrix H
H = cv2.getPerspectiveTransform(src, dst)
Hp = cv2.perspectiveTransform(pts1[None], H)[0]

# Find the inliers by computing the SSD(p',Hp) and saving inliers (feature pa
inliers = []
for i in range(len(pts1)):
ssd = np.sum(np.square(pts2[i] - Hp[i]))
if ssd < epsilon:
inliers.append([pts1[i], pts2[i]])

# Keep the largest set of inliers and the corresponding homography matrix
if len(inliers) > len(best_matches):
best_matches = inliers
```

```
return best_matches

# Plotting tool for convenience


def plot(plots):
fig = plt.figure()
N = len(plots)
x = math.ceil(math.sqrt(N))
y = math.ceil(math.sqrt(N))
for i in range(N):
fig.add_subplot(x, y, i+1).imshow(plots[i])
plt.show()


def mops(img, truth, win_size, h, w, r):
H, W = img.shape
offset = win_size // 2

# Draw line for angle of gradient (for debugging)
#length = 150
#h2 =  int(h - length * math.cos(math.radians(r)))
#w2 =  int(w - length * math.sin(math.radians(r)))
#cv2.line(img, (w, h), (w2, h2), (0,255,0), 2)
```

```
# Rotate image s.t. gradient angle of feature is origin
M = cv2.getRotationMatrix2D((w, h), -1*r, 1)
img_rot = cv2.warpAffine(img, M, (W, H), flags=cv2.INTER_NEAREST)

# Get 40x40 window around feature
win = img_rot[h-offset:h+offset, w-offset:w+offset]

# Size (s,s) of each sample region
s = win_size // 8
# Prefiltering (N,N) -> (N//s, N//s)
i = 0
rows = []
while i < win_size:
j = 0
cols = []
while j < win_size:
# Sample (s,s) region from window
sample = win[i:i+s, j:j+s]
# Downsample (s,s) region to a single value
sample = np.sum(sample) / (s*s)
cols.append(sample)
j += s
rows.append(cols)
i += s

feature = np.array(rows)
```

```
# Normalize
feature = (feature - np.mean(feature)) / np.std(feature)
# Haar wave transform
coeffs = pywt.dwt2(feature, 'haar')
feature = pywt.idwt2(coeffs, 'haar')
#plot([img, img_rot, truth, win, feature])
return feature



def get_matches(ft1, ft2, left_match, right_match, img_left, img_right, max_m
potential_matches = []
offset = win_size // 2

# Feature descriptor with brute-force matching
for h1, w1, r1 in ft1:

# Copy image as to not modify it by reference
img_left_tmp = np.copy(img_left)

# Copy coordinates incase changed by border
h1_tmp = h1
w1_tmp = w1

# Get 40x40 window around feature
win_left = img_left[h1-offset:h1+offset, w1-offset:w1+offset]
```

```
# Handle border points
if win_left.shape != (win_size, win_size):
diff_h, diff_w = np.subtract(win_left.shape, (win_size, win_size))
p_h = abs(diff_h // 2)
p_w = abs(diff_w // 2)
img_left_tmp = cv2.copyMakeBorder(
img_left_tmp, p_h, p_h, p_w, p_w, cv2.BORDER_REFLECT)
h1 += p_h
w1 += p_w
win_left = img_left_tmp[h1-offset:h1+offset, w1-offset:w1+offset]

# Run multiscale oriented patches descriptor
feature_left = mops(img_left_tmp, win_left, win_size, h1, w1, r1)

lowest_dist = math.inf
potential_match = ()
for h2, w2, r2 in ft2:

# Copy image as to not modify it by reference
img_right_tmp = np.copy(img_right)

# Copy coordinates incase changed by border
h2_tmp = h2
w2_tmp = w2

# Get 40x40 window around feature
```

```
win_right = img_right[h2-offset:h2+offset, w2-offset:w2+offset]

# Handle border points
if win_right.shape != (win_size, win_size):
diff_h, diff_w = np.subtract(
win_right.shape, (win_size, win_size))
p_h = abs(diff_h // 2)
p_w = abs(diff_w // 2)
img_right_tmp = cv2.copyMakeBorder(
img_right_tmp, p_h, p_h, p_w, p_w, cv2.BORDER_REFLECT)
h2 += p_h
w2 += p_w
win_right = img_right_tmp[h2 -
offset:h2+offset, w2-offset:w2+offset]

# Run multiscale oriented patches descriptor
feature_right = mops(img_right_tmp, win_right,
win_size, h2, w2, r2)

# Check distance between features
curr_dist = np.linalg.norm(feature_left - feature_right)
if curr_dist < lowest_dist:
lowest_dist = curr_dist
potential_match = ([h1_tmp, w1_tmp, r1], [
h2_tmp, w2_tmp, r2], curr_dist)
```

```
potential_matches.append(potential_match)

# Sort matches from smallest distance up
matches = sorted(potential_matches, key=itemgetter(2))
for match in matches:
# Ensure no duplicates
if match[0][0:2] not in left_match and match[1][0:2] not in right_match:
# Add to matches
left_match.append(match[0][0:2])
right_match.append(match[1][0:2])
# Remove from potential matches
ft1.remove(tuple(match[0]))
ft2.remove(tuple(match[1]))

# Recursively keep going until every point has a match
while(len(left_match) < max_matches and len(right_match) < max_matches):
print('anotha one')
get_matches(ft1, ft2, left_match, right_match,
img_left, img_right, max_matches, win_size)

return np.array(left_match, dtype=np.float32), np.array(right_match, dtype=np



def main(args):
# Load the image
img_left_clr = read_image(args.image1)
```

```
img_right_clr = read_image(args.image2)

# Convert to grayscale
img_left = cv2.cvtColor(img_left_clr, cv2.COLOR_BGR2GRAY)
img_right = cv2.cvtColor(img_right_clr, cv2.COLOR_RGB2GRAY)

print("Getting the features from the Harris Detector")
ftL = harris(img_left, sigma=3, threshold=0.01)
draw_corners(ftL, img_left_clr, 'corners_left')
ftR = harris(img_right, sigma=3, threshold=0.01)
draw_corners(ftR, img_right_clr, 'corners_right')

print("  -- Number of features (left): ", len(ftL))
print("  -- Number of features (right): ", len(ftR))
print("Finding the best matches between images")
with Timer(verbose=True) as t:
max_matches = min(len(ftL), len(ftR))
ptsL, ptsR = get_matches(
ftL, ftR, [], [], img_left, img_right, max_matches, win_size=args.win_size)

print(" -- Number of matches = ", len(ptsL))
assert len(ptsL) == len(ptsR)

print("Performing RANSAC")
matches = ransac(ptsL, ptsR, img_left, img_right,
```

```
args.max_iters, args.epsilon)
print(" -- Number of pruned matches = ", len(matches))

draw_matches(matches, img_left_clr, img_right_clr)



if __name__ == "__main__":
parser = ap.ArgumentParser()
parser.add_argument("image1",
default="NotreDame1.jpg",
type=str,

help="Path to left image")

parser.add_argument("image2",
default="NotreDame2.jpg",
type=str,

help="Path to right image")

parser.add_argument("-w",
"--win_size",
default=50,

type=int,
help="Window size for your feature detector algorithm")
```

```
parser.add_argument("-i",
"--max_iters",
default=1000,

type=int,
help="Maximum iterations to perform RANSAC")

parser.add_argument("-e",
"--epsilon",
default=100,

type=float,
help="SSD epsilon threshold for RANSAC")

parser.add_argument("-v",
"--verbose",
help="Turn on debugging statements",
action="store_true")

args = parser.parse_args()
main(args)


#===================== END of mops.py ==============
```

```
#==================== utils.py ==================
#!/usr/bin/env python3
import numpy as np
import cv2
import time


def read_image(image_path):
img = cv2.imread(image_path)
if img is None:
raise FileNotFoundError("`{}` not cannot be loaded".format(image_path))
return img

def draw_matches(matches, img_left, img_right, verbose=False):
# Determine the max height
height = max(img_left.shape[0], img_right.shape[0])
# Width is the two images side-by-side
width = img_left.shape[1] + img_right.shape[1]

img_out = np.zeros((height, width, 3), dtype=np.uint8)
# Place the images in the empty image
img_out[0:img_left.shape[0], 0:img_left.shape[1], :] = img_left
img_out[0:img_right.shape[0], img_left.shape[1]:, :] = img_right

# The right image coordinates are offset since the image is no longer at (0,0
ow = img_left.shape[1]
```

```
#Draw a line between the matched pairs in green
for p1,p2 in matches:
p1o = (int(p1[1]), int(p1[0]))
p2o = (int(p2[1] + ow), int(p2[0]))
color = list(np.random.random(size=3) * 256)
cv2.line(img_out, p1o, p2o, color, thickness=2)

if verbose:
print("Press enter to continue ... ")
cv2.imshow("matches", img_out)
cv2.waitKey(0)

cv2.imwrite("matches.png", img_out)

class Timer(object):
def __init__(self, verbose=False):
self.verbose = verbose

def __enter__(self):
self.start = time.time()
return self

def __exit__(self, *args):
self.end = time.time()
self.secs = self.end - self.start
```

```
self.mins = self.secs // 60
self.msecs = self.secs * 1000  # millisecs
if self.verbose:
print('elapsed time: {:.2f} s ({:.4f} ms)'.format(self.secs, self.msecs))

#==================== END of utils.py ====================
#==================== harris.py ====================
#!/usr/bin/env python3
import cv2
import numpy as np
import argparse as ap
from utils import read_image


def harris(img, sigma=1, threshold=0.01):
height, width = img.shape
shape = (height, width)
# Calculate the dx,dy gradients of the image (np.gradient doesnt work)
dx = cv2.Sobel(img, cv2.CV_64F,1,0,ksize=5)
dy = cv2.Sobel(img, cv2.CV_64F,0,1,ksize=5)
# Get angle for rotation
_, ang = cv2.cartToPolar(dx, dy, angleInDegrees=True)
# Square the derivatives (A,B,C of H) and apply apply gaussian filters to eac
sigma = (sigma, sigma)
Ixx = cv2.GaussianBlur(dx * dx, sigma, 0)
Ixy = cv2.GaussianBlur(dx * dy, sigma, 0)
```

```
Iyy = cv2.GaussianBlur(dy * dy, sigma, 0)

H = np.array([[Ixx, Ixy], [Ixy, Iyy]])
# Find the determinate
num = (H[0, 0] * H[1, 1]) - (H[0, 1] * H[1, 0])
# Find the trace
denom = H[0,0] + H[1,1]
# Find the response using harmonic mean of the eigenvalues (Brown et. al. var
R = np.nan_to_num(num / denom)

# Adaptive non-maximum suppression, keep the top 1% of values and remove non-
R_flat = R[:].flatten()
# Get number of values in top threshold %
N = int(len(R_flat) * threshold)
# Get values in top threshold %
top_k_percentile = np.partition(R_flat, -N)[-N:]
# Find lowest value in top threshold %
minimum = np.min(top_k_percentile)
# Set all values less than this to 0
R[R < minimum] = 0
# Set non-maximum points in an SxS neighbourhood to 0
s = 9
for h in range(R.shape[0] - s):
for w in range(R.shape[1] - s):
maximum = np.max(R[h:h+s+1, w:w+s+1])
for i in range(h, h+s+1):
```

```
for j in range(w, w+s+1):
if R[i, j] != maximum:
R[i, j] = 0

# Return harris corners in [H, W, R] format
features = list(np.where(R > 0))
features.append(ang[np.where(R > 0)])
corners = zip(*features)
return list(corners)

def draw_corners(corners, img, name):
for h, w, r in corners:
cv2.circle(img, (w, h), 2, (0, 0, 255))

cv2.imwrite(name + '.png', img)
#==================== END of harris.py ============
```

Chapter 3

Answer to the question No 3

3.1 Euclidean distance:

In mathematics, the Euclidean distance or Euclidean metric is the "ordinary" straight-line distance between two points in Euclidean space. With this distance, Euclidean space becomes a metric space. The associated norm is called the Euclidean norm. Older literature refers to the metric as the Pythagorean metric. A generalized term for the Euclidean norm is the L2 norm or L2 distance.

The Euclidean distance between points p and q is the length of the line segment connecting them ($\overline{\mathbf{pqpq}}$).

In Cartesian coordinates, if p = (p1, p2,..., pn) and q = (q1, q2,..., qn) are two points in Euclidean n-space, then the distance (d) from p to q, or from q to p is given by the Pythagorean formula:[1]

$$\begin{aligned} d(\mathbf{p},\mathbf{q}) = d(\mathbf{q},\mathbf{p}) &= \sqrt{(q_1-p_1)^2+(q_2-p_2)^2+\cdots+(q_n-p_n)^2} \\ &= \sqrt{\sum_{i=1}^{n}(q_i-p_i)^2}. \end{aligned} \tag{3.1}$$

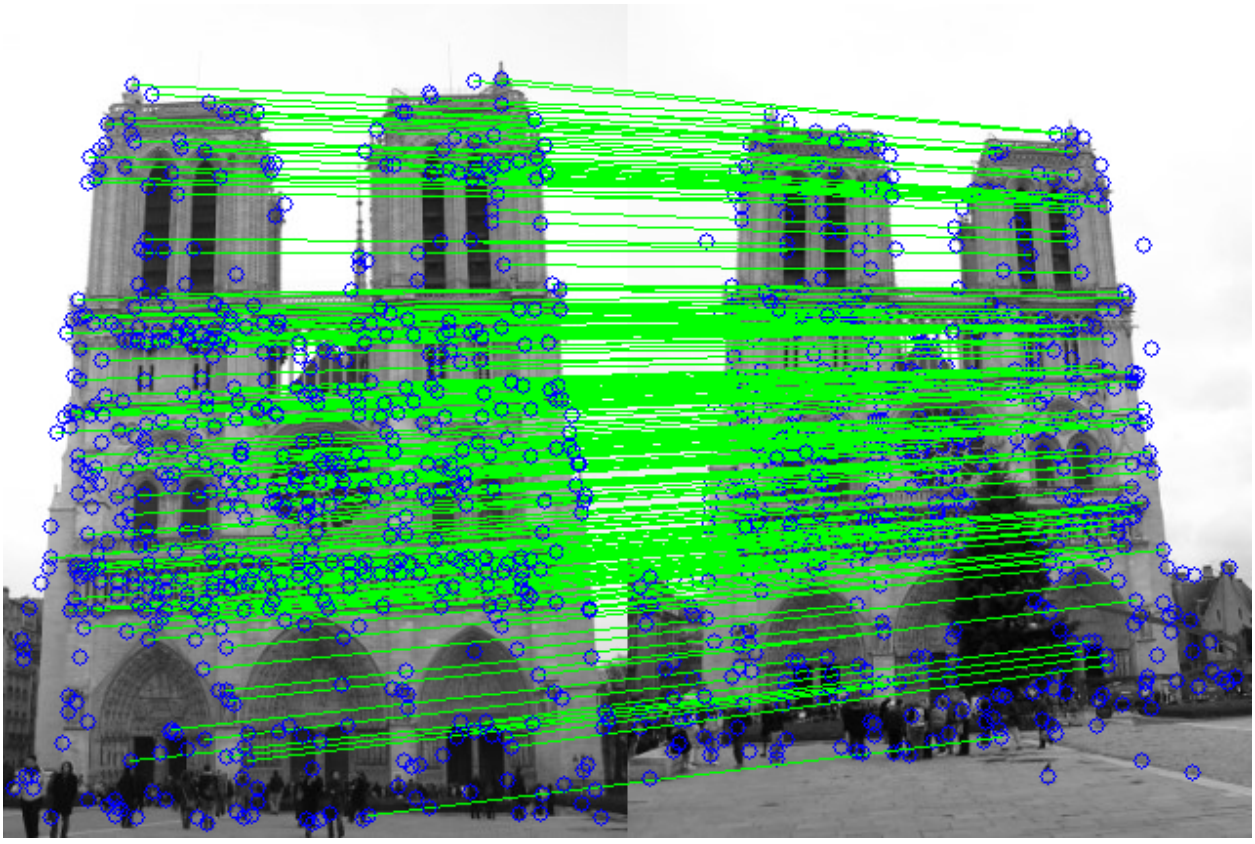


Figure 3.1: Homography Estimation

3.2 Cosine similarity:

Cosine similarity is a measure of similarity between two non-zero vectors of an inner product space that measures the cosine of the angle between them.

The cosine of two non-zero vectors can be derived by using the Euclidean dot product formula:

$$\mathbf{A} \cdot \mathbf{B} = \|\mathbf{A}\|\|\mathbf{B}\| \cos\theta \tag{3.2}$$

Given two vectors of attributes, A and B, the cosine similarity, $\cos\theta$, is represented using a dot product and magnitude as

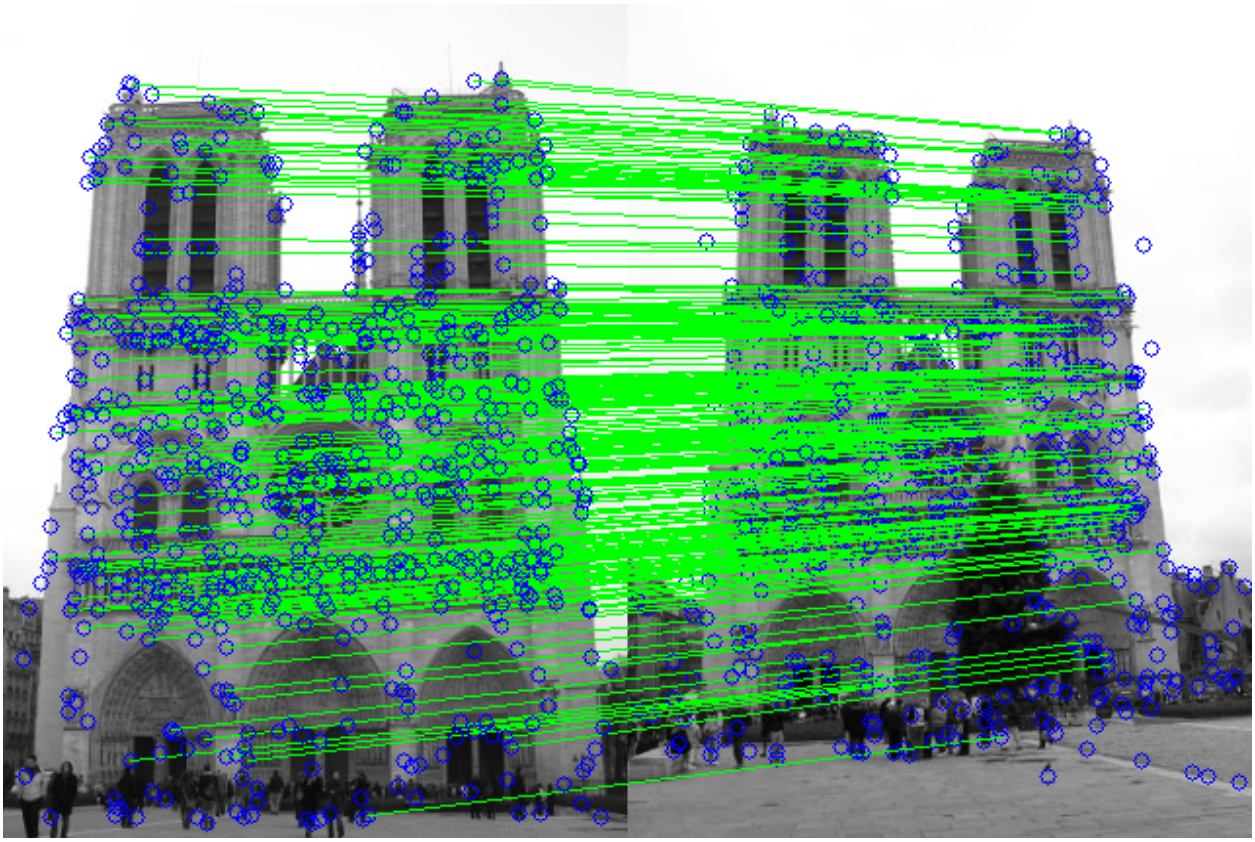


Figure 3.2: Homography Estimation

$$\text{similarity} \ = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \tag{3.3}$$

where A_iA_i and B_iB_i are components of vector AA and BB respectively.

3.3 Corelation:

The correlation between vectors X and Y are defined as follows:

$$r(X,Y) = \frac{\frac{1}{n}\sum_i x_i y_i - \mu_X \mu_y}{\sigma_X \sigma_Y} \tag{3.4}$$

where μ_X and μ_Y are the means of X and Y respectively, and σ_X and σ_Y are the standard deviations of X and Y. The numerator of the equation is called the covariance of X and Y,

and is the difference between the mean of the product of X and Y subtracted from the product of the means. Note that if X and Y are standardized, they will each have a mean of 0 and a standard deviation of 1, so the formula reduces to:

$$r\left(X^*, Y^*\right) = \frac{1}{n}\sum_i x_i y_i \tag{3.5}$$

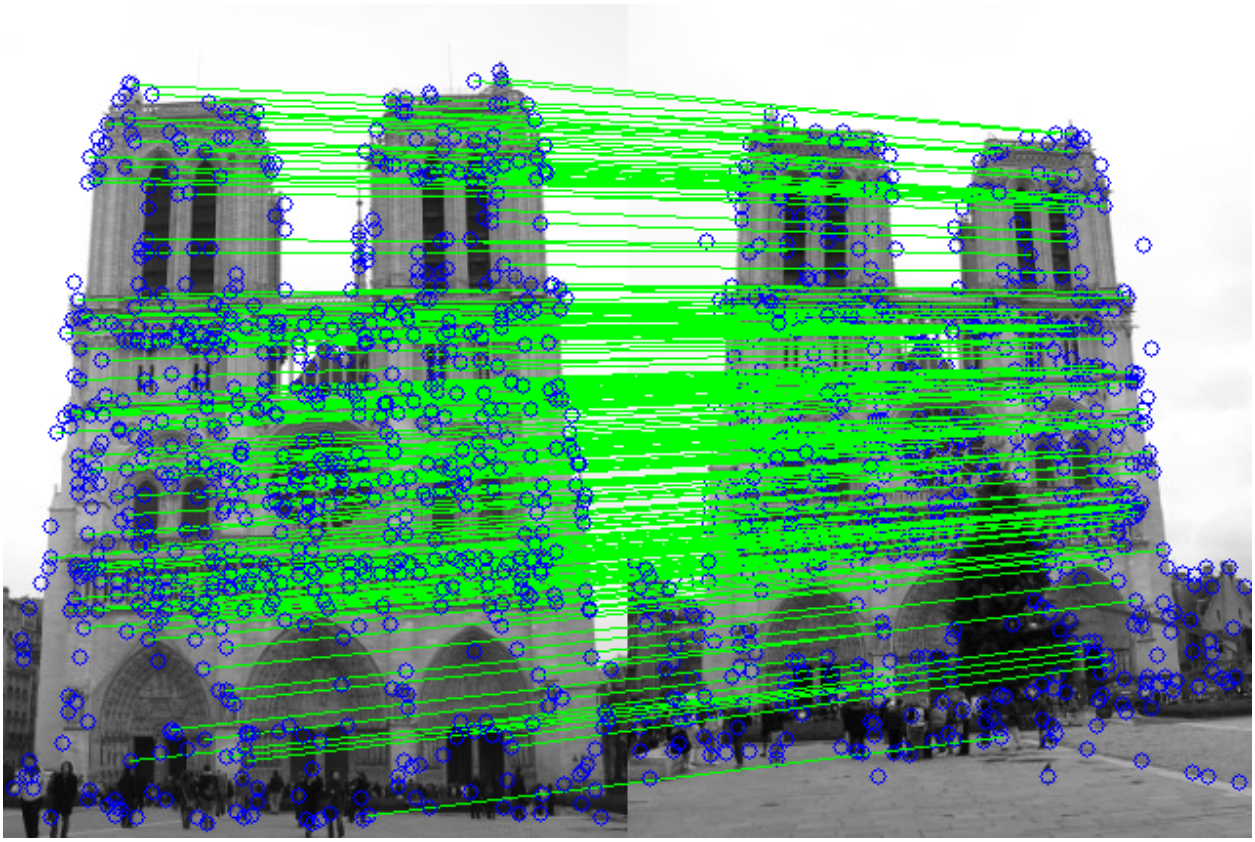


Figure 3.3: Homography Estimation

```
#================================ distance.py ================================
import cv2
import numpy as np
import getopt
```

```
import sys
import random
import matplotlib.pyplot as plt
from scipy import spatial
from scipy.spatial import distance

#
# Read in an image file, errors out if we can't find the file
#


def readImage(filename):
img = cv2.imread(filename, 0)
if img is None:
print('Invalid image:' + filename)
return None
else:
print('Image successfully read...')
return img


def drawMatches(img1, kp1, img2, kp2, matches, inliers=None):
# Create a new output image that concatenates the two images together
rows1 = img1.shape[0]
cols1 = img1.shape[1]
rows2 = img2.shape[0]
```

```
cols2 = img2.shape[1]


out = np.zeros(
(max([rows1, rows2]), cols1+cols2, 3), dtype='uint8')
out2 = np.zeros(
(max([rows1, rows2]), cols1+cols2, 3), dtype='uint8')
out3 = np.zeros(
(max([rows1, rows2]), cols1+cols2, 3), dtype='uint8')
# Place the first image to the left
out[:rows1, :cols1, :] = np.dstack([img1, img1, img1])
out2[:rows1, :cols1, :] = np.dstack([img1, img1, img1])
out3[:rows1, :cols1, :] = np.dstack([img1, img1, img1])

# Place the next image to the right of it
out[:rows2, cols1:cols1+cols2, :] = np.dstack([img2, img2, img2])
out2[:rows2, cols1:cols1+cols2, :] = np.dstack([img2, img2, img2])
out3[:rows2, cols1:cols1+cols2, :] = np.dstack([img2, img2, img2])

# For each pair of points we have between both images
# draw circles, then connect a line between them
for mat in matches:

# Get the matching keypoints for each of the images
img1_idx = mat.queryIdx
img2_idx = mat.trainIdx
```

```
# x - columns, y - rows
(x1, y1) = kp1[img1_idx].pt
(x2, y2) = kp2[img2_idx].pt

inlier = False

if inliers is not None:
for i in inliers:
if i.item(0) == x1 and i.item(1) == y1 and i.item(2) == x2 and i.item(3) == y

inlier = True

dataSetI = [x1, y1]
dataSetII = [x2, y2]

dstcosine = 1 - spatial.distance.cosine(dataSetI, dataSetII)
dsteucla = distance.euclidean(dataSetI, dataSetII)
dstcorelation = distance.correlation(dataSetI, dataSetII)

if dstcosine > 0.90:
# Draw a small circle at both co-ordinates
cv2.circle(out, (int(x1), int(y1)), 4, (255, 0, 0), 1)
cv2.circle(out, (int(x2)+cols1, int(y2)), 4, (255, 0, 0), 1)

# Draw a line in between the two points, draw inliers if we have them
if inliers is not None and inlier:
```

```
cv2.line(out, (int(x1), int(y1)),
(int(x2)+cols1, int(y2)), (0, 255, 0), 1)

if dsteucla > 0.70:
# Draw a small circle at both co-ordinates
cv2.circle(out2, (int(x1), int(y1)), 4, (255, 0, 0), 1)
cv2.circle(out2, (int(x2)+cols1, int(y2)), 4, (255, 0, 0), 1)

# Draw a line in between the two points, draw inliers if we have them
if inliers is not None and inlier:
cv2.line(out2, (int(x1), int(y1)),
(int(x2)+cols1, int(y2)), (0, 255, 0), 1)

if dstcorelation <= 0:
# Draw a small circle at both co-ordinates
cv2.circle(out3, (int(x1), int(y1)), 4, (255, 0, 0), 1)
cv2.circle(out3, (int(x2)+cols1, int(y2)), 4, (255, 0, 0), 1)

# Draw a line in between the two points, draw inliers if we have them
if inliers is not None and inlier:
cv2.line(out3, (int(x1), int(y1)),
(int(x2)+cols1, int(y2)), (0, 255, 0), 1)

return out, out2, out3

#
```

```
# Runs sift algorithm to find features
#


def findFeatures(img):
print("Finding Features...")
# sift = cv2.SIFT()
sift = cv2.xfeatures2d.SIFT_create()
keypoints, descriptors = sift.detectAndCompute(img, None)

return keypoints, descriptors

#
# Matches features given a list of keypoints, descriptors, and images
#


def matchFeatures(kp1, kp2, desc1, desc2, img1, img2):
print("Matching Features...")
matcher = cv2.BFMatcher(cv2.NORM_L2, True)
matches = matcher.match(desc1, desc2)

return matches


#
```

```
# Computers a homography from 4-correspondences
#
def calculateHomography(correspondences):
# loop through correspondences and create assemble matrix
aList = []
for corr in correspondences:
p1 = np.matrix([corr.item(0), corr.item(1), 1])
p2 = np.matrix([corr.item(2), corr.item(3), 1])

a2 = [0, 0, 0, -p2.item(2) * p1.item(0), -p2.item(2) * p1.item(1), -p2.item(2
p2.item(1) * p1.item(0), p2.item(1) * p1.item(1), p2.item(1) * p1.item(2)]
a1 = [-p2.item(2) * p1.item(0), -p2.item(2) * p1.item(1), -p2.item(2) * p1.ite
p2.item(0) * p1.item(0), p2.item(0) * p1.item(1), p2.item(0) * p1.item(2)]
aList.append(a1)
aList.append(a2)

matrixA = np.matrix(aList)

# svd composition
u, s, v = np.linalg.svd(matrixA)

# reshape the min singular value into a 3 by 3 matrix
h = np.reshape(v[8], (3, 3))
# matrixA = np.reshape(matrixA, (3, 3))

# normalize and now we have h
```

```
h = (1/h.item(8)) * h
return h, matrixA



#
# Calculate the geometric distance between estimated points and original poin
#
def geometricDistance(correspondence, h):

p1 = np.transpose(
np.matrix([correspondence[0].item(0), correspondence[0].item(1), 1]))
estimatep2 = np.dot(h, p1)
estimatep2 = (1/estimatep2.item(2))*estimatep2

p2 = np.transpose(
np.matrix([correspondence[0].item(2), correspondence[0].item(3), 1]))
error = p2 - estimatep2
return np.linalg.norm(error)



#
# Runs through ransac algorithm, creating homographies from random correspond
#
def ransac(corr, thresh):
maxInliers = []
finalH = None
```

```
thresh = float(thresh)
for i in range(1000):
# find 4 random points to calculate a homography
corr1 = corr[random.randrange(0, len(corr))]
corr2 = corr[random.randrange(0, len(corr))]
randomFour = np.vstack((corr1, corr2))
corr3 = corr[random.randrange(0, len(corr))]
randomFour = np.vstack((randomFour, corr3))
corr4 = corr[random.randrange(0, len(corr))]
randomFour = np.vstack((randomFour, corr4))

# call the homography function on those points
h, matA = calculateHomography(randomFour)
inliers = []

for i in range(len(corr)):
d = geometricDistance(corr[i], h)
if d < 5:
inliers.append(corr[i])

if len(inliers) > len(maxInliers):
maxInliers = inliers
finalH = h
# print("Corr size: ", len(corr), " NumInliers: ",
#       len(inliers), "Max inliers: ", len(maxInliers))
```

```
if len(maxInliers) > (len(corr)*thresh):
break
return finalH, maxInliers, matA



#
# Main parses argument list and runs the functions
#
def main():

estimation_thresh = 0.60
print("Estimation Threshold: ", estimation_thresh)
if estimation_thresh is None:
estimation_thresh = 0.60

img1name = str("NotreDame1.jpg")
img2name = str("NotreDame2.jpg")
print("Image 1 Name: " + img1name)
print("Image 2 Name: " + img2name)

img1 = readImage("NotreDame1_re.jpg")
img2 = readImage("NotreDame2_re.jpg")

# find features and keypoints
correspondenceList = []
if img1 is not None and img2 is not None:
```

```
kp1, desc1 = findFeatures(img1)
kp2, desc2 = findFeatures(img2)
print("Found keypoints in " + img1name + ": " + str(len(kp1)))
print("Found keypoints in " + img2name + ": " + str(len(kp2)))
keypoints = [kp1, kp2]
matches = matchFeatures(kp1, kp2, desc1, desc2, img1, img2)
for match in matches:
(x1, y1) = keypoints[0][match.queryIdx].pt
(x2, y2) = keypoints[1][match.trainIdx].pt
correspondenceList.append([x1, y1, x2, y2])

corrs = np.matrix(correspondenceList)

# run ransac algorithm
finalH, inliers, matA = ransac(corrs, estimation_thresh)

matchImg, matchImg2, matchImg3 = drawMatches(img1, kp1, img2, kp2,
matches, inliers)

plt.figure(figsize=(20, 20))
plt.imshow(matchImg)
plt.title("Cosine distance")
plt.show()
cv2.imwrite('FinalMatches{}.png'.format(
str("Cosine_distance")), matchImg)
```

```
plt.figure(figsize=(20, 20))
plt.imshow(matchImg2)
plt.title("eucladian distance")
plt.show()
cv2.imwrite('FinalMatches{}.png'.format(
str("eucladian_distance")), matchImg2)

plt.figure(figsize=(20, 20))
plt.imshow(matchImg3)
plt.title("corelation distance")
plt.show()
cv2.imwrite('FinalMatches{}.png'.format(
str("corelation_distance")), matchImg3)


if __name__ == "__main__":
main()
#==================== END of distance.py ==============
```

Chapter 4

Answer to the question No 4

4.1 Showing AH=0 format.

In bellow there's given the homographymatrix in A*H=0 form

$$
A = \left[\begin{array}{ccc}
-3.88850136e+01 & -2.50169159e+02 & -1.00000000e+00 \\
0.00000000e+00 & 0.00000000e+00 & 0.00000000e+00 \\
2.21704490e+03 & 1.42634966e+04 & 5.70154076e+01 \\
\\
0.00000000e+00 & 0.00000000e+00 & 0.00000000e+00 \\
-3.88850136e+01 & -2.50169159e+02 & -1.00000000e+00 \\
8.00298032e+03 & 5.14876728e+04 & 2.05811432e+02 \\
\\
-7.51642914e+01 & -2.93356445e+02 & -1.00000000e+00 \\
0.00000000e+00 & 0.00000000e+00 & 0.00000000e+00 \\
6.89085983e+03 & 2.68941289e+04 & 9.16773071e+01 \\
\\
0.00000000e+00 & 0.00000000e+00 & 0.00000000e+00 \\
-7.51642914e+01 & -2.93356445e+02 & -1.00000000e+00 \\
2.01077852e+04 & 7.84780684e+04 & 2.67517792e+02 \\
\\
-2.54376007e+02 & -2.37659027e+02 & -1.00000000e+00 \\
0.00000000e+00 & 0.00000000e+00 & 0.00000000e+00 \\
5.58146901e+04 & 5.21466827e+04 & 2.19418060e+02 \\
\\
0.00000000e+00 & 0.00000000e+00 & 0.00000000e+00 \\
-2.54376007e+02 & -2.37659027e+02 & -1.00000000e+00 \\
5.69258889e+04 & 5.31848563e+04 & 2.23786392e+02 \\
\\
-6.49763947e+01 & -2.49634460e+02 & -1.00000000e+00 \\
0.00000000e+00 & 0.00000000e+00 & 0.00000000e+00 \\
6.46465207e+03 & 2.48367109e+04 & 9.94923172e+01 \\
\\
\end{array}\right] \tag{4.1}
$$

$$A(lastpart) = \begin{bmatrix} 0.00000000e+00 & 0.00000000e+00 & 0.00000000e+00 \\ -6.49763947e+01 & -2.49634460e+02 & -1.00000000e+00 \\ 1.49495965e+04 & 5.74352343e+04 & 2.30077347e+02 \end{bmatrix} \tag{4.2}$$

$$*H = \begin{bmatrix} 9.14199645e-01 & 2.98561459e-02 & 5.93901311e+01 \\ 9.12617813e-02 & 8.22202292e-01 & 7.54071582e+01 \\ 7.76337462e-05 & -2.33285995e-05 & 1.00000000e+00 \end{bmatrix} \tag{4.3}$$

$$= \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{4.4}$$

so this is the representation from of A * H = 0

Chapter 5

Answer to the question No 5

5.1 Homography

Typically, homographies are estimated between images by finding feature correspondences in those images.

$$H = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \tag{5.1}$$

Let (x_1, y_1) be a point in the first image and (x_2, y_2) be the coordinates of the same physical point in the second image. Then, the Homography H relates them in the following way

$$\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = H \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} \tag{5.2}$$

Two images of a scene are related by a homography under two conditions.As mentioned earlier, a homography is nothing but a 3x3 matrix as shown below.

Two images of a scene are related by a homography under two conditions.

- The two images are that of a plane (e.g. sheet of paper, credit card etc.).
- The two images were acquired by rotating the camera about its optical axis. We take such images while generating panoramas

Under homography, we can write the transformation of points in 3D from camera 1 to camera 2 as:

$$\mathbf{X}_2 = H\mathbf{X}_1 \quad \mathbf{X}_1, \mathbf{X}_2 \in \mathbb{R}^3 \tag{5.3}$$

In the image planes, using homogeneous coordinates, we have $\lambda_1\mathbf{x}_1 = \mathbf{X}_1, \quad \lambda_2\mathbf{x}_2 = \mathbf{X}_2$, therefore $\lambda_2\mathbf{x}_2 = H\lambda_1\mathbf{x}_1$ This means that x2 is equal to Hx1 up to a scale (due to universal scale ambiguity). $\mathbf{x}_2 \sim H\mathbf{x}_1$ is a direct mapping between points in the image planes. If it is known that some points all lie in a plane in an image1 , the image can be rectified directly without needing to recover and manipulate 3D coordinates.

To estimate H, we start from the equation x2 $\sim$ Hx1. Written element by element, in homogenous coordinates we get the following constraint:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \sim \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{5.4}$$

In inhomogenous coordinates

$$x' = \frac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + h_{33}} \tag{5.5}$$

$$y' = \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + h_{33}} \tag{5.6}$$

One approach can be set $\mathbf{h}_{33} = \mathbf{1}$:

$$x' = \frac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + (1)} \tag{5.7}$$

$$y' = \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + (1)} \tag{5.8}$$

Second approach can be impose unit vector constraint

$$x' = \frac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + h_{33}} \tag{5.9}$$

$$y' = \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + h_{33}} \tag{5.10}$$

Subject to the constraint

$$h_{11}^2 + h_{12}^2 + h_{13}^2 + h_{21}^2 + h_{22}^2 + h_{23}^2 + h_{31}^2 + h_{32}^2 + h_{33}^2 = 1 \tag{5.11}$$

L.S. using Algebraic Distance. Multiplying through by denominator from equation 3.7,3.8

$$(h_{31}x + h_{32}y + 1)\, x' = h_{11}x + h_{12}y + h_{13} \tag{5.12}$$

$$(h_{31}x + h_{32}y + 1)\, x' = h_{11}x + h_{12}y + h_{13} \tag{5.13}$$

By rearrange

$$h_{11}x + h_{12}y + h_{13} - h_{31}xx' - h_{32}yx' = x' \tag{5.14}$$

$$h_{21}x + h_{22}y + h_{23} - h_{31}xy' - h_{32}yy' = y' \tag{5.15}$$

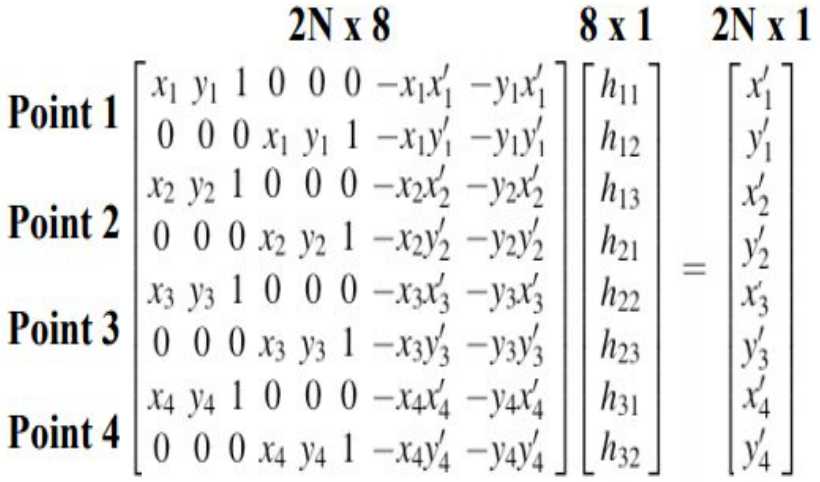


Figure 5.1: Homography calculation

Given a set of corresponding points, we can form the following linear system of equations,

$$A\mathbf{h} = \mathbf{0} \tag{5.16}$$

Equation can be solved using homogeneous linear least squares.known as the Homogeneous Linear Least Squares problem. It is similar in appearance to the inhomogeneous linear least squares problem

$$A\mathbf{x} = \mathbf{b} \tag{5.17}$$

in which case we solve for x using the pseudoinverse or inverse of A. This won't work with Equation 15. Instead we solve it using Singular Value Decomposition (SVD). Starting with equation 13 from the previous section, we first compute the SVD of A

$$A = U\Sigma V^{\top} = \sum_{i=1}^{9} \sigma_i \mathbf{u}_i \mathbf{v}_i^{\top} \tag{5.18}$$

When performed in Matlab, the singular values σ_i will be sorted in descending order, so σ_9 will be the smallest. There are three cases for the value of σ_9

- If the homography is exactly determined, then $\sigma_9 = 0$, and there exists a homography that fits the points exactly.
- If the homography is overdetermined, then $\sigma_9 \geq 0$. Here σ_9 represents a "residual" or goodness of fit.
- We will not handle the case of the homography being underdetermined.

From the SVD we take the "right singular vector" (a column from V) which corresponds to the smallest singular value, σ_9. This is the solution, h, which contains the coefficients of the homography matrix that best fits the points. We reshape h into the matrix H, and form the equation x2 $\sim$ Hx1.

$$\begin{array}{cccc} \text{2Nx8} & \text{8xl} & & \text{2Nx1} \\ \text{A} & \text{h} & = & \text{b} \end{array} \tag{5.19}$$

$$\begin{array}{cccc} \overset{\text{8x2N}}{\mathbf{A}^{\mathbf{T}}} & \overset{\text{2Nx8}}{\mathbf{A}} & \overset{\text{8x1}}{\mathbf{h}} & = \quad \overset{\text{8x2N}}{\mathbf{A}^{\mathbf{T}}} \quad \overset{\text{2Nx1}}{\mathbf{b}} \\ \overbrace{(\mathbf{A}^{\mathbf{T}} \quad \mathbf{A})}^{\text{8x8}} & & \overset{\text{8x1}}{\mathbf{h}} & = \quad \overbrace{(\mathbf{A}^{\mathbf{T}} \quad \mathbf{b})}^{\text{8x1}} \end{array}$$

Figure 5.2: Homography Estimation

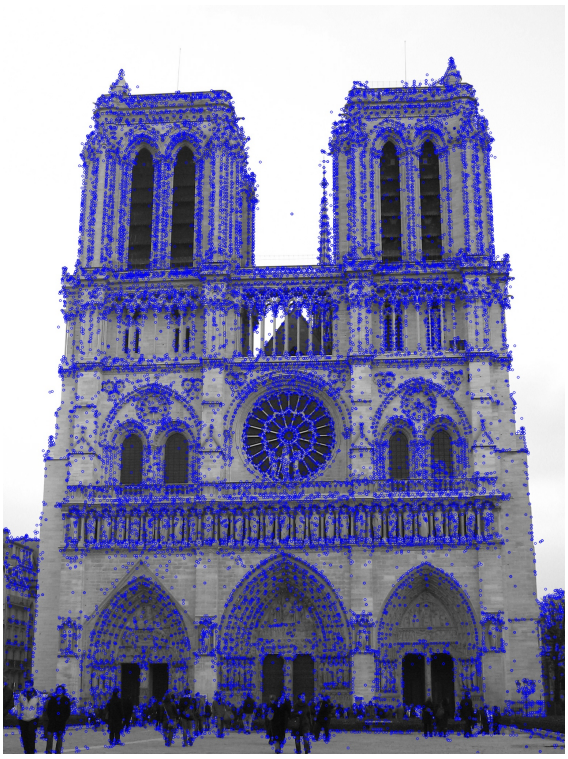
Finally

$$\mathbf{h} = \left(\mathbf{A}^{\mathrm{T}}\mathbf{A}\right)^{-1}\left(\mathbf{A}^{\mathrm{T}} \quad \mathbf{b}\right) \tag{5.20}$$

So this way we can calculate the homography matrix H.

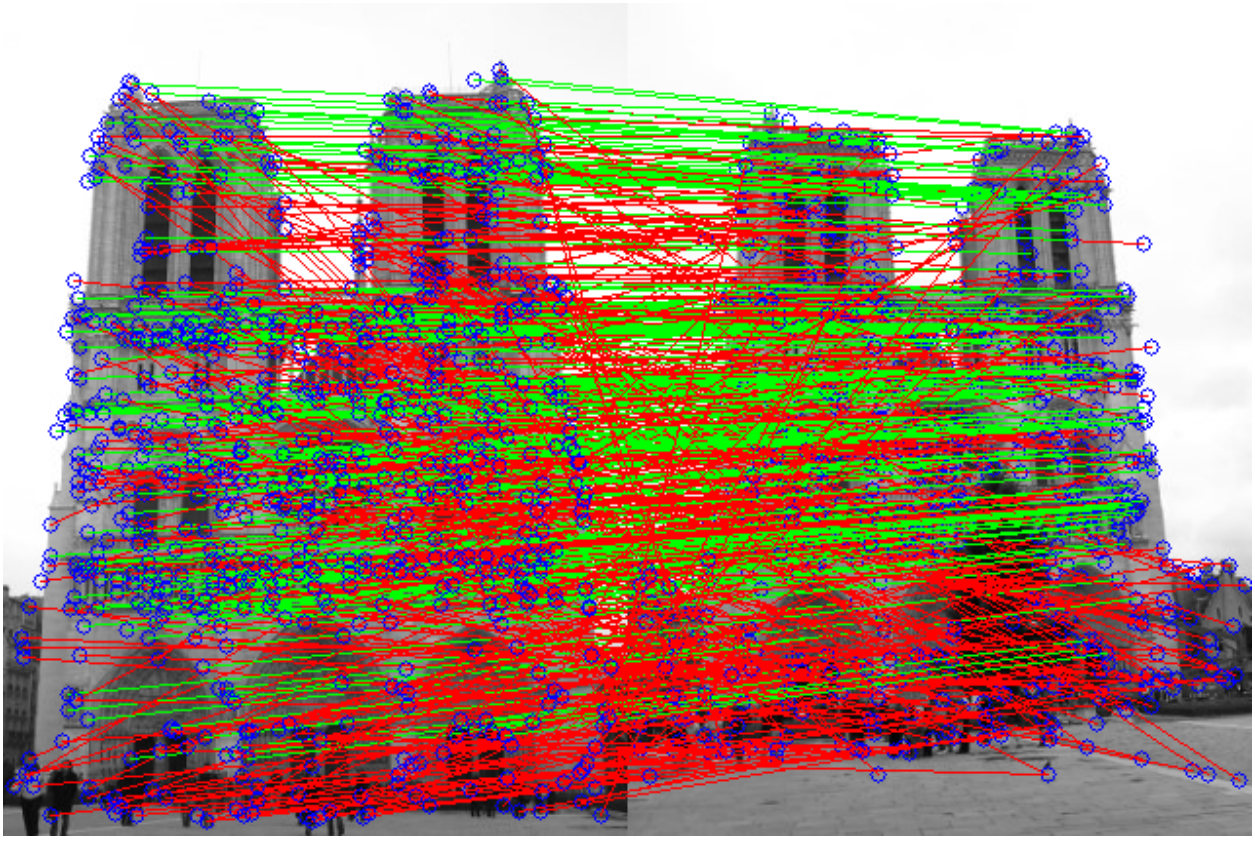
Here is a python implementation of Homography Estimation:

#========== homography . py ==========



(a) Sift key point detected First image

(b) Sift key point detected Second image



(c) Key point matching: RED:not perfect match, GREEN:Fully match

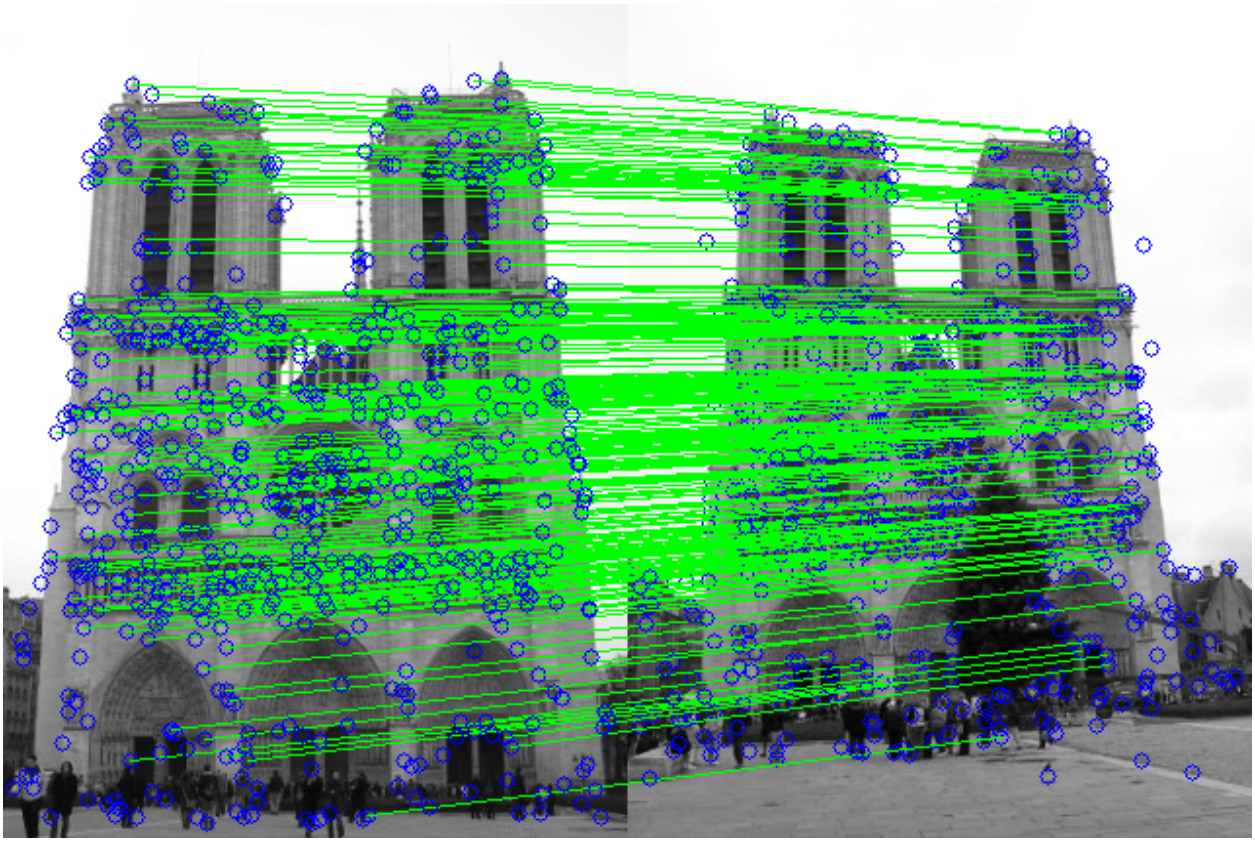


Figure 5.4: Removing unwanted outline with RANSAC and finnaly mateched image

```
import cv2
import numpy as np
import getopt
import sys
import random

#
# Read in an image file, errors out if we can't find the file
#
```

```
def readImage(filename):
img = cv2.imread(filename, 0)
if img is None:
print('Invalid image:' + filename)
return None
else:
print('Image successfully read...')
return img


def drawMatches(img1, kp1, img2, kp2, matches, inliers=None):
# Create a new output image that concatenates the two images together
rows1 = img1.shape[0]
cols1 = img1.shape[1]
rows2 = img2.shape[0]
cols2 = img2.shape[1]

out = np.zeros((max([rows1, rows2]), cols1+cols2, 3), dtype='uint8')

# Place the first image to the left
out[:rows1, :cols1, :] = np.dstack([img1, img1, img1])

# Place the next image to the right of it
out[:rows2, cols1:cols1+cols2, :] = np.dstack([img2, img2, img2])
```

```
# For each pair of points we have between both images
# draw circles, then connect a line between them
for mat in matches:

# Get the matching keypoints for each of the images
img1_idx = mat.queryIdx
img2_idx = mat.trainIdx

# x - columns, y - rows
(x1, y1) = kp1[img1_idx].pt
(x2, y2) = kp2[img2_idx].pt

inlier = False

if inliers is not None:
for i in inliers:
if i.item(0) == x1 and i.item(1) == y1 and i.item(2) == x2 and i.item(3) == y
inlier = True

# Draw a small circle at both co-ordinates
cv2.circle(out, (int(x1), int(y1)), 4, (255, 0, 0), 1)
cv2.circle(out, (int(x2)+cols1, int(y2)), 4, (255, 0, 0), 1)

# Draw a line in between the two points, draw inliers if we have them
if inliers is not None and inlier:
```

```
cv2.line(out, (int(x1), int(y1)),
(int(x2)+cols1, int(y2)), (0, 255, 0), 1)
elif inliers is not None:
cv2.line(out, (int(x1), int(y1)),
(int(x2)+cols1, int(y2)), (0, 0, 255), 1)

if inliers is None:
cv2.line(out, (int(x1), int(y1)),
(int(x2)+cols1, int(y2)), (255, 0, 0), 1)

return out

#
# Runs sift algorithm to find features
#


def findFeatures(img):
print("Finding Features...")
#sift = cv2.SIFT()
sift = cv2.xfeatures2d.SIFT_create()
keypoints, descriptors = sift.detectAndCompute(img, None)

img = cv2.drawKeypoints(img, keypoints, outImage=None)
cv2.imwrite('sift_keypoints.png', img)
```

```
return keypoints, descriptors

#
# Matches features given a list of keypoints, descriptors, and images
#


def matchFeatures(kp1, kp2, desc1, desc2, img1, img2):
print("Matching Features...")
matcher = cv2.BFMatcher(cv2.NORM_L2, True)
matches = matcher.match(desc1, desc2)
matchImg = drawMatches(img1, kp1, img2, kp2, matches)
cv2.imwrite('Matches.png', matchImg)
return matches



#
# Computers a homography from 4-correspondences
#
def calculateHomography(correspondences):
# loop through correspondences and create assemble matrix
aList = []
for corr in correspondences:
p1 = np.matrix([corr.item(0), corr.item(1), 1])
p2 = np.matrix([corr.item(2), corr.item(3), 1])
```

```
a2 = [0, 0, 0, -p2.item(2) * p1.item(0), -p2.item(2) * p1.item(1), -p2.item(2
p2.item(1) * p1.item(0), p2.item(1) * p1.item(1), p2.item(1) * p1.item(2)]
a1 = [-p2.item(2) * p1.item(0), -p2.item(2) * p1.item(1), -p2.item(2) * p1.ite
p2.item(0) * p1.item(0), p2.item(0) * p1.item(1), p2.item(0) * p1.item(2)]
aList.append(a1)
aList.append(a2)

matrixA = np.matrix(aList)

# svd composition
u, s, v = np.linalg.svd(matrixA)

# reshape the min singular value into a 3 by 3 matrix
h = np.reshape(v[8], (3, 3))
# matrixA = np.reshape(matrixA, (3, 3))

# normalize and now we have h
h = (1/h.item(8)) * h
return h, matrixA



#
# Calculate the geometric distance between estimated points and original poin
#
def geometricDistance(correspondence, h):
```

```
p1 = np.transpose(
np.matrix([correspondence[0].item(0), correspondence[0].item(1), 1]))
estimatep2 = np.dot(h, p1)
estimatep2 = (1/estimatep2.item(2))*estimatep2

p2 = np.transpose(
np.matrix([correspondence[0].item(2), correspondence[0].item(3), 1]))
error = p2 - estimatep2
return np.linalg.norm(error)



#
# Runs through ransac algorithm, creating homographies from random correspond
#
def ransac(corr, thresh):
maxInliers = []
finalH = None
thresh = float(thresh)
for i in range(1000):
# find 4 random points to calculate a homography
corr1 = corr[random.randrange(0, len(corr))]
corr2 = corr[random.randrange(0, len(corr))]
randomFour = np.vstack((corr1, corr2))
corr3 = corr[random.randrange(0, len(corr))]
randomFour = np.vstack((randomFour, corr3))
corr4 = corr[random.randrange(0, len(corr))]
```

```
randomFour = np.vstack((randomFour, corr4))

# call the homography function on those points
h, matA = calculateHomography(randomFour)
inliers = []

for i in range(len(corr)):
d = geometricDistance(corr[i], h)
if d < 5:
inliers.append(corr[i])

if len(inliers) > len(maxInliers):
maxInliers = inliers
finalH = h
print("Corr size: ", len(corr), " NumInliers: ",
len(inliers), "Max inliers: ", len(maxInliers))

if len(maxInliers) > (len(corr)*thresh):
break
return finalH, maxInliers, matA



#
# Main parses argument list and runs the functions
#
def main():
```

```
# args, img_name = getopt.getopt(sys.argv[1:], '', ['threshold='])
# args = dict(args)

# estimation_thresh = args.get('--threshold')
estimation_thresh = 0.60
print("Estimation Threshold: ", estimation_thresh)
if estimation_thresh is None:
estimation_thresh = 0.60

img1name = str("NotreDame1_re.jpg")
img2name = str("NotreDame2_re.jpg")
print("Image 1 Name: " + img1name)
print("Image 2 Name: " + img2name)

## query image
# img1 = readImage(img_name[0])
## train image
# img2 = readImage(img_name[1])

img1 = readImage("NotreDame1_re.jpg")
img2 = readImage("NotreDame2_re.jpg")

# find features and keypoints
correspondenceList = []
if img1 is not None and img2 is not None:
kp1, desc1 = findFeatures(img1)
```

```
kp2, desc2 = findFeatures(img2)
print("Found keypoints in " + img1name + ": " + str(len(kp1)))
print("Found keypoints in " + img2name + ": " + str(len(kp2)))
keypoints = [kp1, kp2]
matches = matchFeatures(kp1, kp2, desc1, desc2, img1, img2)
for match in matches:
(x1, y1) = keypoints[0][match.queryIdx].pt
(x2, y2) = keypoints[1][match.trainIdx].pt
correspondenceList.append([x1, y1, x2, y2])

corrs = np.matrix(correspondenceList)

# run ransac algorithm
finalH, inliers, matA = ransac(corrs, estimation_thresh)

print("A = {} \n ".format(matA))
print("H = {} \n \n".format(finalH))
print("{} * {} = 0 \n".format(matA, finalH))
print("Final homography: ", finalH)
print("Final inliers count: ", len(inliers))

matchImg = drawMatches(img1, kp1, img2, kp2, matches, inliers)
cv2.imwrite('InlierMatches.png', matchImg)

f = open('homography.txt', 'w')
f.write("Final homography: \n \n" + str(finalH)+"\n")
```

```
f.write("Final inliers count: \n" + str(len(inliers))+"\n")
f.write(""+str("{} * {} = 0".format(matA, finalH)))
f.close()


if __name__ == "__main__":
main()
```

#========== END of homography.py ==========

Chapter 6

Answer to the question No 6

6.1 Applying H to transform first image

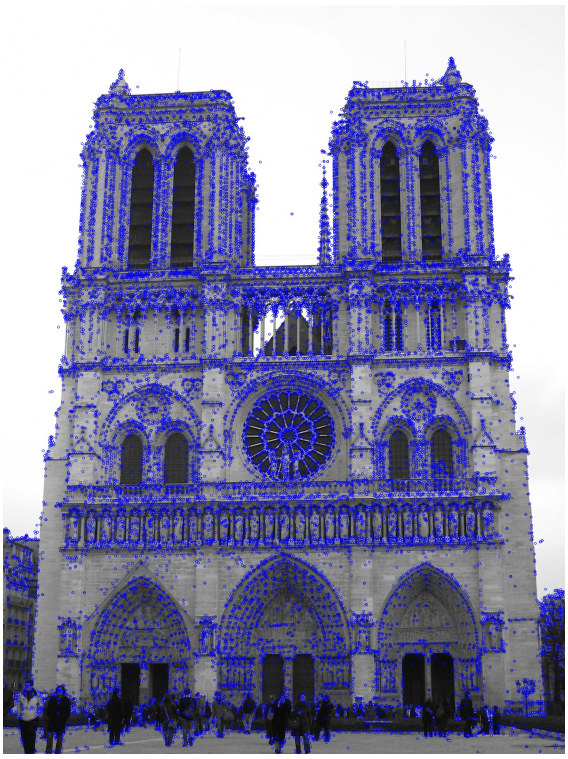
According to the question first we take the two picture and compute their homography matrix. Then apply the homography matrix H to the first image.In bellow the value of H(Hmogarphy matrix is given).

$$H = \begin{bmatrix} 9.14199645e-01 & 2.98561459e-02 & 5.93901311e+01 \\ 9.12617813e-02 & 8.22202292e-01 & 7.54071582e+01 \\ 7.76337462e-05 & -2.33285995e-05 & 1.00000000e+00 \end{bmatrix} \tag{6.1}$$

Here is a python implementation of :

```
#==================== homography_applied_image ====================
```

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
```



(a) Sift key point detected First image

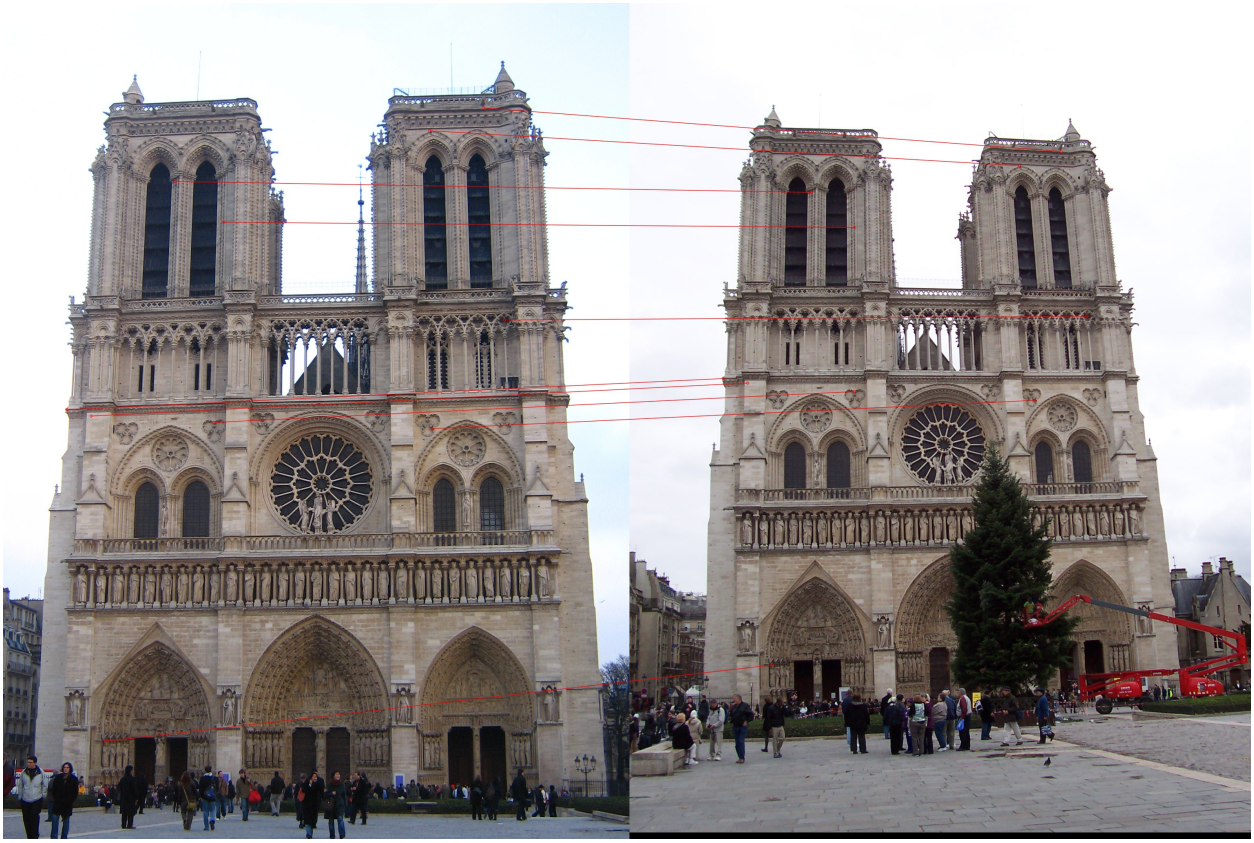
(b) Sift key point detected Second image

(c) Key point matching between the two image

$$H = \begin{bmatrix} 9.14199645e-01 & 2.98561459e-02 & 5.93901311e+01 \\ 9.12617813e-02 & 8.22202292e-01 & 7.54071582e+01 \\ 7.76337462e-05 & -2.33285995e-05 & 1.00000000e+00 \end{bmatrix}$$

(a) Homography matrix

(b) Homography applied image



(c) Taking only 10 point for homography calculation

Figure 6.2: Homography matrix calculation and apply

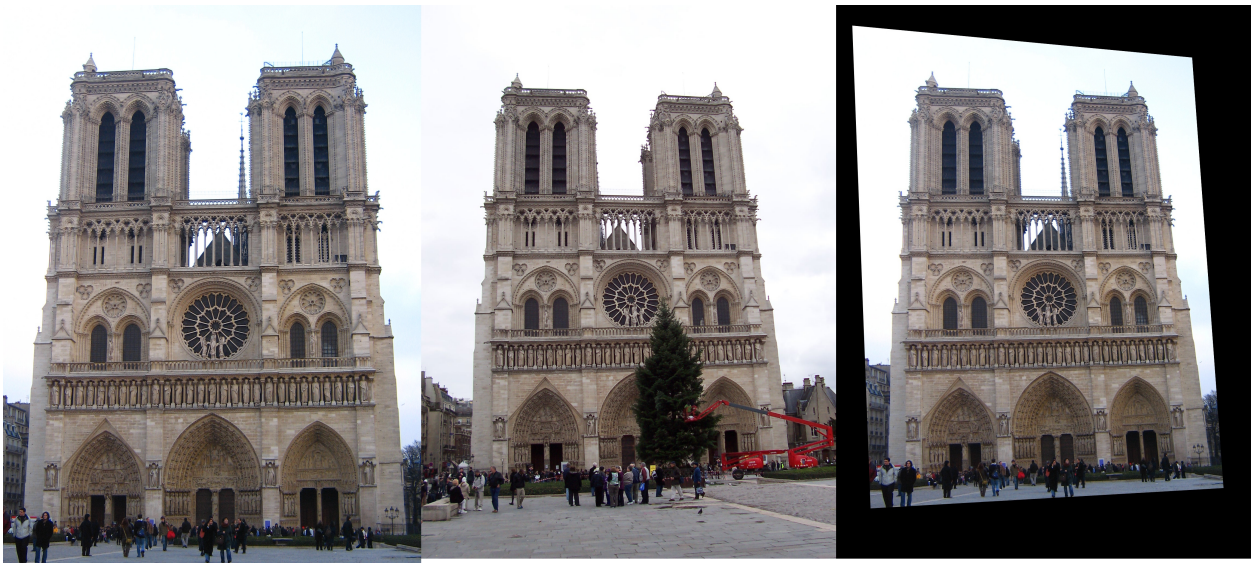


Figure 6.3: left: 1st image, middle: 2nd image, right:H applied image [here H matrix is applied in first image]

```
import math



# Importing the images
pic1 = cv2.imread("NotreDame1.jpg")
pic2 = cv2.imread("NotreDame2.jpg")



# Converting images to gray-scale
graypic1 = cv2.cvtColor(pic1, cv2.COLOR_BGR2GRAY)
graypic2 = cv2.cvtColor(pic2, cv2.COLOR_BGR2GRAY)


# Creating SIFT object
sift = cv2.xfeatures2d.SIFT_create()
```

```
# Finding keypoints and descriptors for Mountain images
keyp1, desc1 = sift.detectAndCompute(graypic1, None)
keyp2, desc2 = sift.detectAndCompute(graypic2, None)

# Drawing keypoints for Mountain images
keyImage1 = cv2.drawKeypoints(graypic1, keyp1, np.array([]), (255, 0, 0)
keyImage2 = cv2.drawKeypoints(graypic2, keyp2, np.array([]), (255, 0, 0)

cv2.imwrite('sift1.jpg', keyImage1)
cv2.imwrite('sift2.jpg', keyImage2)



# Brute-Force matching with SIFT descriptors
brutef = cv2.BFMatcher()

# Matching the keypoints with k-nearest neighbor (with k=2)
matches = brutef.knnMatch(desc1, desc2, k=2)

goodMatch = []
# Performing ratio test to find good matches
for m, n in matches:
if m.distance < 0.75*n.distance:
goodMatch.append(m)

# Drawing good matches
matchImage = cv2.drawMatches(
```

```
pic1, keyp1, pic2, keyp2, goodMatch, np.array([]), (255, 0, 255), flags=

cv2.imwrite('matches_knn.jpg', matchImage)


# Getting source and destination points
srce_pts = np.float32(
[keyp1[m.queryIdx].pt for m in goodMatch]).reshape(-1, 1, 2)
dest_pts = np.float32(
[keyp2[m.trainIdx].pt for m in goodMatch]).reshape(-1, 1, 2)

# Finding Homography Matrix and mask
homographyMat, mask = cv2.findHomography(srce_pts, dest_pts, cv2.RANSAC,
print(homographyMat)


# Converting the mask to a list
matchesMask = mask.ravel().tolist()

h, w = pic1.shape[:2]
pts = np.float32([[0, 0], [0, h-1], [w-1, h-1], [w-1, 0]]).reshape(-1, 1

matchIndex = []
for i in range(len(matchesMask)):
if (matchesMask[i]):
matchIndex.append(i)
```

```
matchArray = []
for i in matchIndex:
matchArray.append(goodMatch[i])

# Finding 10 random matches using inliers
np.random.seed(sum([ord(c) for c in UBIT]))
randomMatch = np.random.choice(matchArray, 10, replace=False)

# Defining draw parameters
draw_params = dict(matchColor=(0, 0, 255),
singlePointColor=None,
flags=2)

# Drawing the match image for 10 random points
matchImage = cv2.drawMatches(
pic1, keyp1, pic2, keyp2, randomMatch, None, **draw_params)

cv2.imwrite('matches.jpg', matchImage)



h1, w1 = pic2.shape[:2]
h2, w2 = pic1.shape[:2]
pts1 = np.float32([[0, 0], [0, h1], [w1, h1], [w1, 0]]).reshape(-1, 1, 2
pts2 = np.float32([[0, 0], [0, h2], [w2, h2], [w2, 0]]).reshape(-1, 1, 2
pts2_ = cv2.perspectiveTransform(pts2, homographyMat)
```

```
pts = np.concatenate((pts1, pts2_), axis=0)

# Finding the minimum and maximum coordinates
[xmin, ymin] = np.int32(pts.min(axis=0).ravel() - 0.5)
[xmax, ymax] = np.int32(pts.max(axis=0).ravel() + 0.5)
t = [-xmin, -ymin]

# Translating
Ht = np.array([[1, 0, t[0]], [0, 1, t[1]], [0, 0, 1]])

# Warping the first image on the second image using Homography Matrix
result = cv2.warpPerspective(pic1, Ht.dot(
homographyMat), (xmax-xmin, ymax-ymin))


cv2.imwrite('Homography_applied_image.jpg', result)

#==================== END of homography_applied_image.py ==============
```