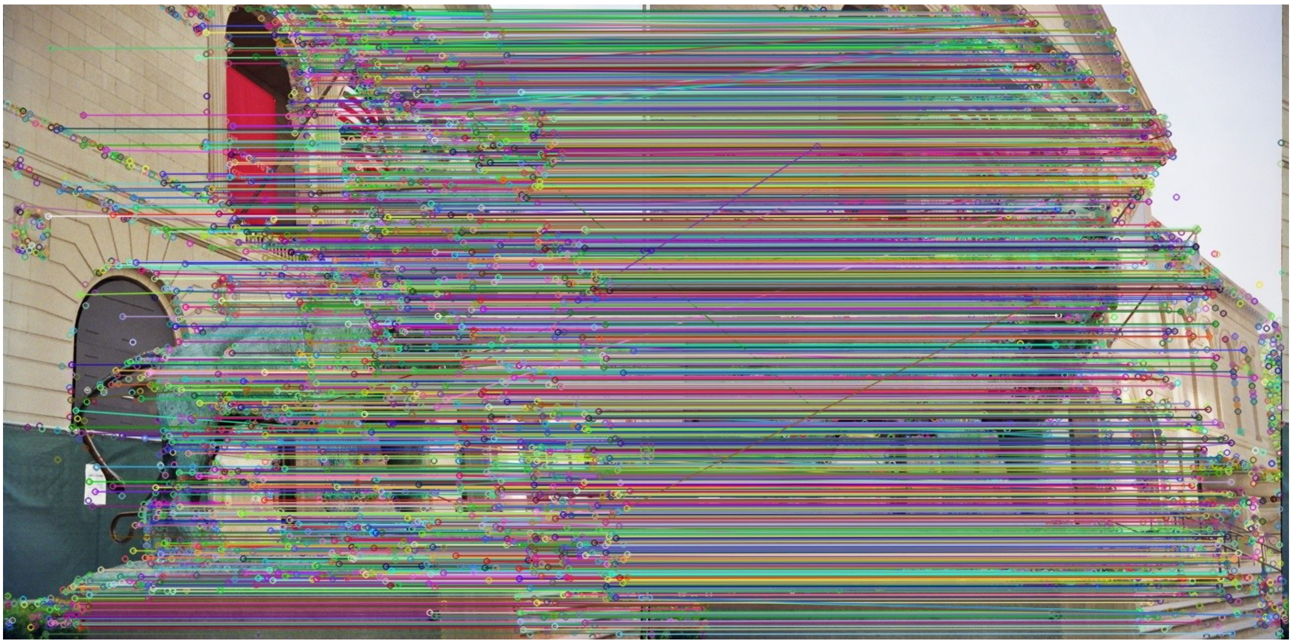
Assignement 3:
Udit Singh Parihar
Roll number - 2018701024

1. Dense Sift
a.



b.

c.

2.

3. Correlation method contains many outliers.

4.

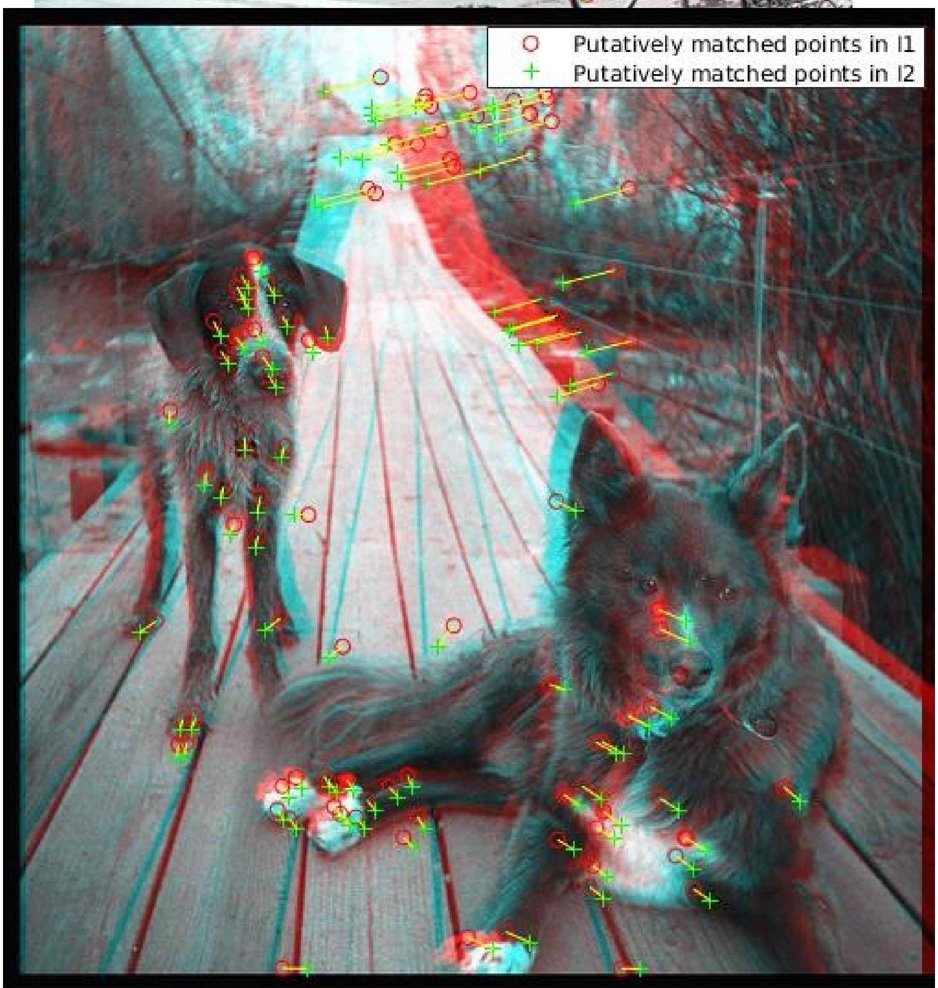
a. Before Rectification

After Rectification: Correspondences lie on a horizontal line

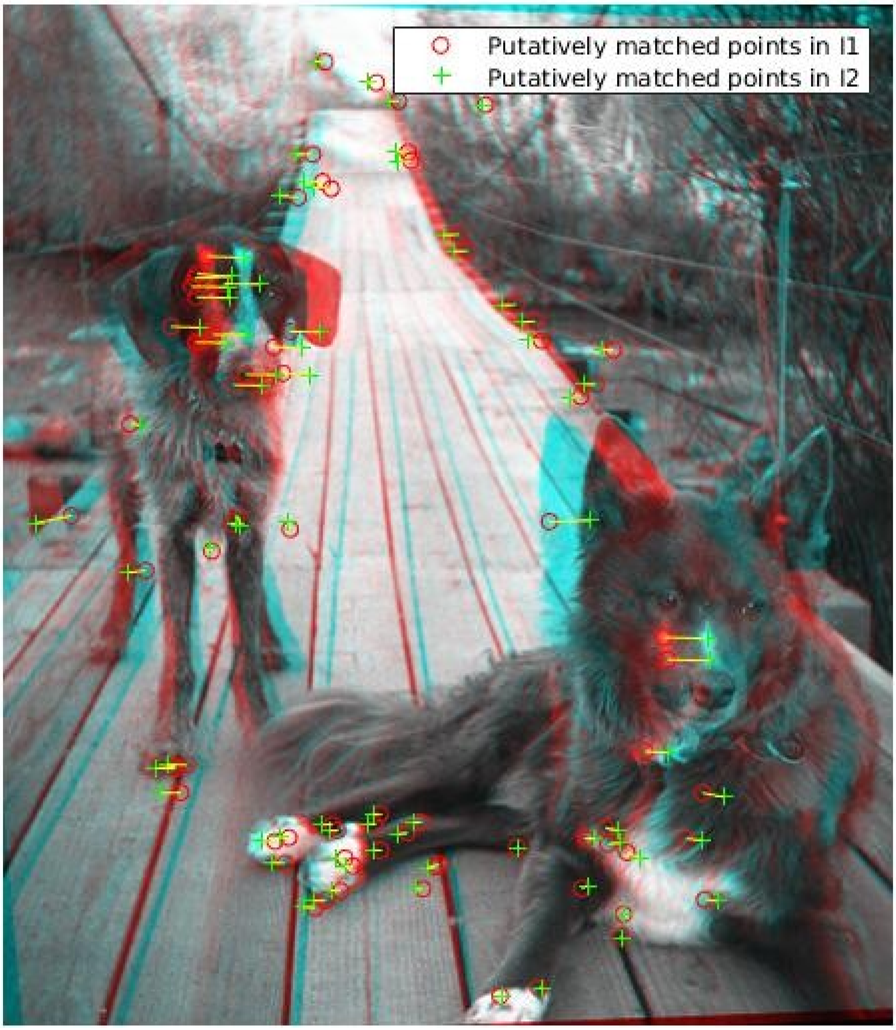
b. Before Rectification

After Rectification: Correspondences lie on a horizontal parallel line

c. Before Rectification



After Rectification : Correspondences move to parallel horizontal lines.



Codes:

// 1. sift_matcher.cpp – Dense sift

```
#include <iostream>
#include <opencv2/opencv.hpp>
#include <opencv2/xfeatures2d.hpp>

using namespace std;
using namespace cv;
```

```
void display_image(const Mat& image){
	namedWindow("opencv_viewer", WINDOW_AUTOSIZE);
	imshow("opencv_viewer", image);
	waitKey(0);
	destroyWindow("opencv_viewer");
}

void save_image(const Mat& image, string image_name){
	image_name += ".jpg";
	imwrite(image_name, image);
	cout << image_name << " saved in current directory.\n";
}

void rescale_image(const Mat& in_image, Mat& out_image, float scale){
	resize(in_image, out_image, Size(in_image.cols/scale, in_image.rows/scale));
}

void split_image(const Mat3b& rgb, Mat3b& new_rgb, int offset){
	for(int y=0; y<new_rgb.rows; ++y){
		for(int x=0; x<new_rgb.cols; ++x){
			new_rgb.at<Vec3b>(y, x) = rgb.at<Vec3b>(y, x+offset);
		}
	}
}

void apply_sift(const Mat& rgb1, const Mat& rgb2){
	Mat descriptors1, descriptors2;
	vector<KeyPoint> keypoints1, keypoints2;
	const int features_count = 500000;
	Ptr<xfeatures2d::SIFT> feature_detector = xfeatures2d::SIFT::create(features_count);
	feature_detector->detectAndCompute(rgb1, noArray(), keypoints1, descriptors1);
	feature_detector->detectAndCompute(rgb2, noArray(), keypoints2, descriptors2);

	vector<vector<DMatch>> knn_matches;
	Ptr<DescriptorMatcher> matcher = 
DescriptorMatcher::create(DescriptorMatcher::FLANNBASED);
	matcher->knnMatch(descriptors1, descriptors2, knn_matches, 2);

	vector<DMatch> matches;
	const float threshold_ratio = 0.7f;
	for(size_t i=0; i<knn_matches.size(); ++i){
		if(knn_matches[i][0].distance < threshold_ratio * knn_matches[i][1].distance)
			matches.push_back(knn_matches[i][0]);
	}

	Mat final_image;
	drawMatches(rgb1, keypoints1, rgb2, keypoints2, matches, final_image);
	save_image(final_image, "dense_sift");
	display_image(final_image);
}

int main(int argc, char const *argv[]){
```

```
	if(argc != 2){
		fprintf(stdout, "Usage: %s rgb.png\n", argv[0]);
		return 1;
	}
	Mat rgb = imread(argv[1], IMREAD_COLOR );
	if(rgb.empty()){
		fprintf(stdout, "Unable to open image\n");
		return 1;
	}

	const float scale = 1.1;
	rescale_image(rgb, rgb, scale);

	Size new_size = Size(rgb.rows, rgb.cols/2);
	Mat3b rgb1(new_size, CV_8UC3);
	Mat3b rgb2(new_size, CV_8UC3);
	split_image(rgb, rgb1, 0);
	split_image(rgb, rgb2, rgb.cols/2);

	display_image(rgb);
	display_image(rgb1);
	display_image(rgb2);

	apply_sift(rgb1, rgb2);

	return 0;
}

//2.  window_matching.cpp – Correlation based window matching

#include <iostream>
#include <opencv2/opencv.hpp>
#include <opencv2/xfeatures2d.hpp>
#include <opencv2/imgproc.hpp>

using namespace std;
using namespace cv;

void display_image(const Mat& image){
	namedWindow("opencv_viewer", WINDOW_AUTOSIZE);
	imshow("opencv_viewer", image);
	waitKey(0);
	destroyWindow("opencv_viewer");
}

void rescale_image(const Mat& in_image, Mat& out_image, float scale){
	resize(in_image, out_image, Size(in_image.cols/scale, in_image.rows/scale));
}

void split_image(const Mat& rgb, Mat& new_rgb, int offset){
	for(int y=0; y<new_rgb.rows; ++y){
		for(int x=0; x<new_rgb.cols; ++x){
```

```
				new_rgb.at<Vec3b>(y, x) = rgb.at<Vec3b>(y, x+offset);
			}
		}
}

void process_image(Mat& rgb, Mat& norm1, Mat& norm2){
		const float scale = 1.7;
		rescale_image(rgb, rgb, scale);

		Size new_size = Size(rgb.rows, rgb.cols/2);
		Mat3b rgb1(new_size, CV_8UC3);
		Mat3b rgb2(new_size, CV_8UC3);
		split_image(rgb, rgb1, 0);
		split_image(rgb, rgb2, rgb.cols/2);

		Mat img1(rgb1.size(), CV_8U);
		Mat img2(rgb2.size(), CV_8U);
		cvtColor(rgb1, img1, COLOR_RGB2GRAY);
		cvtColor(rgb2, img2, COLOR_RGB2GRAY);

		normalize(img1, norm1, 1, -1, NORM_MINMAX, CV_32F);
		normalize(img2, norm2, 1, -1, NORM_MINMAX, CV_32F);
}

void fill_window(const Mat& norm1, const Point& kp, Mat& window){
		const int row = window.rows, col = window.cols;
		for(int y=kp.y-(row/2), i=0; i<row; ++y, ++i){
			for(int x=kp.x-(col/2), j=0; j<col; ++x, ++j){
				window.at<float>(i, j) = norm1.at<float>(y, x);
			}
		}
}

void corresponding_window(const Mat& norm2, const Mat& window, Point& keypoint2){
		Mat convolve_norm;
		Point anchor(0, 0);
		const int ddepth = -1, delta = 0;
		filter2D(norm2, convolve_norm, ddepth, window, anchor, delta, BORDER_REPLICATE);

		double min_value, max_value;
		Point min_point, max_point;
		minMaxLoc(convolve_norm, &min_value, &max_value, &min_point, &max_point);
		keypoint2.x = max_point.x+window.cols/2;
		keypoint2.y = max_point.y+window.rows/2;
}

int main(int argc, char const *argv[]){
		if(argc != 2){
			fprintf(stdout, "Usage: %s rgb.png\n", argv[0]);
			return 1;
		}
		Mat rgb = imread(argv[1], IMREAD_COLOR );
```

```
if(rgb.empty()){
	fprintf(stdout, "Unable to open image\n");
	return 1;
}

Mat norm1, norm2;
// process_image(rgb, norm1, norm2);
const float scale = 1.7;
rescale_image(rgb, rgb, scale);

Size new_size = Size(rgb.rows, rgb.cols/2);
Mat3b rgb1(new_size, CV_8UC3);
Mat3b rgb2(new_size, CV_8UC3);
split_image(rgb, rgb1, 0);
split_image(rgb, rgb2, rgb.cols/2);

Mat img1(rgb1.size(), CV_8U);
Mat img2(rgb2.size(), CV_8U);
cvtColor(rgb1, img1, COLOR_RGB2GRAY);
cvtColor(rgb2, img2, COLOR_RGB2GRAY);

normalize(img1, norm1, 1, -1, NORM_MINMAX, CV_32F);
normalize(img2, norm2, 1, -1, NORM_MINMAX, CV_32F);

vector<Point> coord1, coord2;
const int win_size=15;
Mat window(win_size, win_size, CV_32F);
int count = 0, total=(norm1.rows * norm1.cols);

for(int i=2*win_size; i<norm1.rows-2*win_size ; ++i){
	for(int j=2*win_size; j<norm1.cols-2*win_size; ++j){
		Point keypoint1(i, j), keypoint2;
		fill_window(norm1, keypoint1, window);
		corresponding_window(norm2, window, keypoint2);
		coord1.push_back(keypoint1);
		coord2.push_back(keypoint2);

		if(count%1000 == 0)
			cout << "Count: " << count << "/" << total << endl;
		++count;
	}
}

Mat F, H1, H2;
F = findFundamentalMat(coord1, coord2, CV_FM_RANSAC);
stereoRectifyUncalibrated(coord1, coord2, F, norm1.size(), H1, H2);

Mat warped_rgb1, warped_rgb2;
cv::Size warped_image_size(norm1.cols*2, norm1.rows);
warpPerspective(rgb1, warped_rgb1, H1, warped_image_size);
warpPerspective(rgb2, warped_rgb2, H2, warped_image_size);
```

```
        display_image(warped_rgb1);
        display_image(warped_rgb2);

        return 0;
}
```

// 3. uncalibrated_rectify.m – Stereo rectification

```
I = imread('data/stereo_images/Stereo_Pair3.jpg');
[row, col, chan] = size(I);
I1 = imcrop(I, [1,1,floor(col/2),row]);
I2 = imcrop(I, [ceil(col/2), 1, floor(col/2), row]);

I1gray = rgb2gray(I1);
I2gray = rgb2gray(I2);

blobs1 = detectSURFFeatures(I1gray, 'MetricThreshold', 2000);
blobs2 = detectSURFFeatures(I2gray, 'MetricThreshold', 2000);

[features1, validBlobs1] = extractFeatures(I1gray, blobs1);
[features2, validBlobs2] = extractFeatures(I2gray, blobs2);
indexPairs = matchFeatures(features1, features2, 'Metric', 'SAD','MatchThreshold', 5);
matchedPoints1 = validBlobs1(indexPairs(:,1),:);
matchedPoints2 = validBlobs2(indexPairs(:,2),:);
matchedPoints1(1)

figure;
showMatchedFeatures(I1, I2, matchedPoints1, matchedPoints2);
legend('Putatively matched points in I1', 'Putatively matched points in I2');

[fMatrix, epipolarInliers, status] = estimateFundamentalMatrix(...
  matchedPoints1, matchedPoints2, 'Method', 'RANSAC', ...
  'NumTrials', 10000, 'DistanceThreshold', 0.1, 'Confidence', 99.99);

inlierPoints1 = matchedPoints1(epipolarInliers, :);
inlierPoints2 = matchedPoints2(epipolarInliers, :);

[t1, t2] = estimateUncalibratedRectification(fMatrix, ...
  inlierPoints1.Location, inlierPoints2.Location, size(I2));
tform1 = projective2d(t1);
tform2 = projective2d(t2);

[I1Rect, I2Rect] = rectifyStereoImages(I1, I2, tform1, tform2);
figure;
imshow(I1Rect);
figure;
imshow(I2Rect);

I1gray = rgb2gray(I1Rect);
I2gray = rgb2gray(I2Rect);

blobs1 = detectSURFFeatures(I1gray, 'MetricThreshold', 2000);
```

```
blobs2 = detectSURFFeatures(I2gray, 'MetricThreshold', 2000);

[features1, validBlobs1] = extractFeatures(I1gray, blobs1);
[features2, validBlobs2] = extractFeatures(I2gray, blobs2);
indexPairs = matchFeatures(features1, features2, 'Metric', 'SAD','MatchThreshold', 5);
matchedPoints1 = validBlobs1(indexPairs(:,1),:);
matchedPoints2 = validBlobs2(indexPairs(:,2),:);

figure;
showMatchedFeatures(I1Rect, I2Rect, matchedPoints1, matchedPoints2);
legend('Putatively matched points in I1', 'Putatively matched points in I2');
```