pyImageStitching

By: Alessandro Ferrari

email: alessandroferrari87@gmail.com

Linux Day LugBS

26 October 2013, Brescia

I. INTRODUCTION

The goal of this todays lab is to develop with open source instruments an image stitching pipeline for creating panorama images. The lab has been implemented in python language, using OpenCV and numpy libraries. The image stitching is performed by using a cascade of a features detector module (sift), a features matching module (flann) and an inlier pair matches estimation for select the best matches (ransac).

This approach to image stitching can be slower and less accurate compared to older approaches, however the results are much more robust to noise, lighting gradients and no requirements are asked about relative position between images.

Alternative to SIFT other features module, such as surf, orb, brisk, kaze and so on, can be used alike to sift in the implementation of the pipeline.
The code for the experiment can be cloned by git at the repository https://github.com/alessandroferrari/pyImageStitching.git. The simulation can be run simply by launching from the terminal "python image_stitching.py". Have fun!

II. SIFT- SCALE INVARIANT FEATURES TRANSFORM

SIFT implements invariance to scale by means of a gaussian pyramid scale space. It implements also affine transformation invariance and lighting gradients invariant. The SIFT implementation used in teh code is offered by OpenCV library. The code for computing SIFT features on an image is:

```
detector = cv2.SIFT(nfeatures=0, nOctaveLayers=3, contrastThreshold=0.04, edgeThreshold=10, sigma=1.6)
kp1, desc1 = self.detector.detectAndCompute(img1, None)
```

By using the parameters in input at the features detector constructor, it is possible to tune the desired results. If nfeatures is equal to zero is determined according to the other parameters, otherwise is forced to its value. contrastThreshold controls the strength of the features detected. The higher its value, the stronger has to be a feature to be detected. sigma parameter determines the sigma of gaussian kernel used to smoothing in the scale space.

kp1 indicates the coordinates of the points where the features have been detected. desc1 indicates the descriptors relative to those points. Descriptors are used to perform matches between the features of different images.

III. FLANN - FAST LINEAR APPROXIMATED NEAREST NEIGHBOURS

Each feature detected in an image has a descriptor that describes its radial and intensity characteristics normalized with respect to scale space and orientation. This is fundamental for implementing panorama pictures using the previously briefly introduced pipeline. In fact, features of the same objects in two different images, are paired by matching features descriptors. The challenge to pair multidimensional descriptors (as sift descriptors are), it is called to find the nearest neighbours. Exact nearest neighbours do not scale well with the increase of dimensionality of the descriptors and numbers of features. So an approximate approach is chosen for trading off accuracy if benefit to speed.

FLANN library selects the most suitable approximate nearest neighbours algorithm for the dataset where matching as to be performed. However, features matching is not robust by itself. Several pairs are wrong and they would compromise the image stitching process. This issue will be addressed with the next step of the pipeline. The code to implement flann follows:

```
FLANN_INDEX_KDTREE = 1  # opencv bug: flann enums are missing
norm = cv2.NORM_L2  # with different norms parameters may change
flann_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
```

```
matcher = cv2.FlannBasedMatcher(flann_params, {})
raw_matches = self.matcher.knnMatch(desc1, trainDescriptors = desc2, k = 2)
p1, p2, kp_pairs = self.filter_matches(kp1, kp2, raw_matches)
```

Flann matcher is initialized. kp1 and desc1 are the features in the first image and its relative descriptors. kp2 and desc2 are the features in the second image and its relative descriptors. knnMatch method of the matcher find the pair of best neighbours and the second best matching, since k is equal to 2. The method filter matches discards weak pairs, in fact if the best matching is not way better than the second one, it is higher the probability of a false match. So, all the matchings where the first pair is not greater enough compared to the second match are discarded. kp pairs contains the pairs of features. It follows the code of filter matches:

```
def filter_matches(kp1, kp2, matches, ratio = 0.75):
    mkp1, mkp2 = [], []
    for m in matches:
        if len(m) == 2 and m[0].distance < m[1].distance * ratio:
            m = m[0]
            mkp1.append( kp1[m.queryIdx] )
            mkp2.append( kp2[m.trainIdx] )
    p1 = np.float32([kp.pt for kp in mkp1])
    p2 = np.float32([kp.pt for kp in mkp2])
    kp_pairs = zip(mkp1, mkp2)
    return p1, p2, kp_pairs
```

IV. RANSAC - RANDOM SAMPLE CONSENSUS ALGORITHM

Some features pairs are correct, while some others are completely wrong. If the image homograhy designed to perform overlapping take into account also the incorrect pairs, the final result will be affected by error. In order to have acceptable panorama pictures, such error cannot be accepted, since even really small displacement results are really tedious to sight. So, ransac is an algorithm that select a subset of inliers pairs matching, while discarding outliers matching. Inliers are pairs that are "coherent" with the homography for overlapping, while outliers do not fit into it. Consider a dataset of N features pairs, ransac select a smaller subset of M features pairs, calculate the model according to them. After calculated the homography, it establishes the number of features pairs that are closer than a certain tolerance, the inliers. If the number of inliers is bigger than a number K, it found a suitable solution, otherwise it will try a new solution selecting a new subset of M samples and repeating the previous steps. In this way, outliers are discarded and only best matches are kept to perform the overlapping. Just 4 good matches are necessary to find a correct homography, once that ransac method is performed, the homography is computed.

It follows the code for ransac and homography computation in python and OpenCV:

```
if len(p1) >= 4:
    H, status = cv2.findHomography(p1, p2, cv2.RANSAC, 5.0)
```

V. IMAGES BLENDING AND OVERLAPPING

Once that the homography matrix is computed, the homography can be applied to the source image, so that it will overlap to the target image. For doing so, it is possible simply to use the cv2.warpPerspective function in OpenCV library. However, this is not enough in order to obtain good panorama images. In fact there may be strong differences of light due to the fact that exposure times of the shutters and white balance control of the cameras changes when capturing the same scene by different perspectives. So it is necessary to perform images blending, in order to make smoother the discontinuities between the two overlapped pictures.

It follows the code of the functions used to perform the overlapping:

```
#given the height and the width of the panorama, and the barrier, that indicates
#where there is the discontinuity between the images, this function produce
#a smoothed transient in the overlapping.
#smoothing window is a parameter that determines the width of the transient
#left_biased is a flag that determines whether it is masked the left image,
#or the right one
def blending_mask(height, width, barrier, smoothing_window, left_biased=True):

    assert barrier < width

    mask = np.zeros((height, width))

    offset = int(smoothing_window/2)
    if left_biased:
        mask[:,barrier-offset:barrier+offset+1]=np.tile(np.linspace(1,0,2*offset+1).T, (height, 1))
        mask[:,:barrier-offset] = 1
    else:
        mask[:,barrier-offset:barrier+offset+1]=np.tile(np.linspace(0,1,2*offset+1).T, (height, 1))
        mask[:,barrier+offset:] = 1

    return cv2.merge([mask, mask, mask])
```

```
#this function apply the homography to img2 and it performs the blending while doing so
def images_blending(img1, img2, width_panorama, height_panorama, H, smoothing_window = 400):

    barrier = img1.shape[1] -int(smoothing_window/2)

    panorama1 = np.zeros((height_panorama, width_panorama, 3))
    mask1 = blending_mask(height_panorama, width_panorama, barrier, smoothing_window = smoothing_window,
        left_biased = True)
    panorama1[0:img1.shape[0],0:img1.shape[1],:] = img1
    panorama1 *= mask1

    mask2 = blending_mask(height_panorama, width_panorama, barrier, smoothing_window = smoothing_window,
        left_biased = False)
    panorama2 = cv2.warpPerspective(img2, H, (width_panorama, height_panorama)) * mask2

    return panorama1 + panorama2
```

```
#apply the homography for overlapping images, meanwhile performing blending between
#them
panorama_image = images_blending(img1, img2, width_panorama, height_panorama, H)
```

VI. RESULTS

Some visual results of the experiment follows.

I hope you have enjoyed it!