Review of Common Mistakes in Lab 1

transform_homgraphy() [1%]

This part is easy, and most people get full credit. However, many students use for-loops to implement this part instead of much-more-efficient numpy operations. Here are some example code snippets on how to implement this part efficiently.

To augment the third coordinate to obtain homogeneous representation:

```
src = np.pad(src, ((0, 0), (0, 1)), mode='constant', constant_values=1.0)
```

To transform all the homogeneous coordinates:

Notice that you can transform all coordinates at one go by transposing it into a $3 \times N$ matrix and pre-multiplying it by $\mathbf{H}$.

```
transformed = (h_matrix @ src.transpose()).transpose()
```

compute_homography() [4%]

Most students have no problems formulating the $\mathbf{Ax} = \mathbf{0}$ system of equations (Well done!). Common problems are 1) Not normalizing, and 2) Bug in normalization code.

Normalization is essential for stability and I penalize severely if it is not done. Do remember to normalize for the 8-point algorithm in Assignment 3.

warp_image() [2%]

This part is causing more problems that I expected. The suggested `cv2.remap()` handles a large part of the complexity for you, but many students do not know how to use it correctly. Many students end up using their own check for masking (e.g. by checking whether the pixel is completely dark) which unfortunately is flawed.

Here's some code snippets to implement the important parts efficiently:

To generate the coordinates to transform:

```
height_dst, width_dst = dst.shape[:2]
xy, yv = np.meshgrid(np.arange(width_dst, dtype=np.float64),
                     np.arange(height_dst, dtype=np.float64),
                     indexing='xy')
xy_dst = np.stack([xv, yv], axis=-1)
xy_dst_flattened = xy_dst.reshape((-1, 2))
```

To perform the warping:

```
dst = cv2.remap(src, x_map, y_map, interpolation=cv2.INTER_LINEAR, dst=dst,
                borderMode=cv2.BORDER_TRANSPARENT)
```

Notice that you **pass `dst` into the function**. Pixels which are not overwritten will retain the values from dst.

compute_homography_error () / compute_homography_ransac() [5%]

Common errors for this function are:

1. Not using minimal subset: Estimating homography requires just **4** correspondences. By using more than the minimal subset, the number of trials needed dramatically increases.

2. Did not re-estimate homography with all detected inliers: It is important to use all inliers to improve the accuracy of your solution.

3. RANSAC sometimes crashes: Many students pick repeated points when performing RANSAC, and the function crashes when all 4 correspondences are the same. This is because your normalizing function in `compute_homography()` will give a division-by-zero. You should sample **without replacement**, which can be done by using either of the following:

```
subset = np.random.choice(num_pairs, size=4, replace=False)
# or
import random
subset = random.sample(range(num_pairs), 4)
```

concatenate_homographies() [3%]

Only half the students get full credit for this part. Most of those who are wrong multiplied the matrices in the wrong direction.

Notice that the provided pairwise homography matrices have the following convention:

$$\mathbf{x}_{k+1} = \mathbf{H}_k^{k+1}\mathbf{x}_k,$$

and the instructions ask for all the absolute homography matrices $\mathbf{H}_i^{ref}$.

Therefore, for images $i < ref$,

$$\mathbf{H}_i^{ref} = \mathbf{H}_{ref-1}^{ref}\mathbf{H}_{ref-2}^{ref-1} \dots \mathbf{H}_{i+1}^{i+2}\mathbf{H}_i^{i+1},$$

and for images $i > ref$,

$$\begin{aligned}\mathbf{H}_i^{ref} &= \mathbf{H}_{ref+1}^{ref}\mathbf{H}_{ref+2}^{ref+1} \dots \mathbf{H}_{i-1}^{i-2}\mathbf{H}_i^{i-1} \\ &= \left(\mathbf{H}_{ref}^{ref+1}\right)^{-1}\left(\mathbf{H}_{ref+1}^{ref+2}\right)^{-1} \dots \left(\mathbf{H}_{i-2}^{i-1}\right)^{-1}\left(\mathbf{H}_{i-1}^{i}\right)^{-1}\end{aligned}$$