2018-17119 박수현

Explanation of HW3 code

Part 1 Homography

`H = compute_h(p1, p2)`

Given that this function must return a 3 x 3 homography matrix, I set h to be the flatten-out 9 x 1 matrix and define a least squares problem $\|Ah - 0\|^2$. 2N x 9 matrix A is constructed using the coordinates of each correspondence in p1 and p2, and singular value decomposition $\mathrm{A} = \mathrm{U}\Sigma\mathrm{V}^{\mathrm{T}}$ is performed. Since I have proven in Question 2 of the theory part of the homework that the *last column of* V is the solution $\hat{h}$, I retrieve the *last row of* V^T obtained by the SVD. Then the output H matrix is reshaped into a 3 x 3 form.

`H = compute_h_norm(p1, p2)`

A normalization step can improve the numerical stability of the solution, to be more stable in presence of noise and to prevent divergence from the correct result.

I followed the data normalization technique explained by Rafael Garcia[1], where the method was originally proposed by Hartley and Zisserman[2].

First, the coordinates in each image are translated so that the centroid of the points is set as the origin of coordinates.

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n} \quad \text{and} \quad \bar{y} = \frac{\sum_{i=1}^{n} y_i}{n}$$

Next, the coordinates are scaled so that the average distance from a point to the origin is $\sqrt{2}$. The initial average distance from every point to the origin of coordinates is calculated as follows:

$$\bar{d} = \frac{\sum_{i=1}^{n} \sqrt{(x_i - \bar{x})^2 + (y_i - \bar{y})^2}}{n}$$

Now the scaling factor can be computed as $s = \frac{\sqrt{2}}{\bar{d}}$. As a result, the translation and scaling can be performed by means of the transformation matrix T.

$$\mathbf{T} = \begin{pmatrix} \frac{\sqrt{2}}{\bar{d}} & 0 & -\left(\frac{\sqrt{2}}{\bar{d}}\bar{x}\right) \\ 0 & \frac{\sqrt{2}}{\bar{d}} & -\left(\frac{\sqrt{2}}{\bar{d}}\bar{y}\right) \\ 0 & 0 & 1 \end{pmatrix}$$

I derived two translation matrices T1 and T2, each for p1 and p2. Through matrix multiplication, p1 and p2 are normalized, which will be passed to compute_H() and get the initial homography matrix. The standard homography matrix can be obtained through the following denormalization equation:

$$\mathrm{H} = (T1)^{-1} H (T2)$$

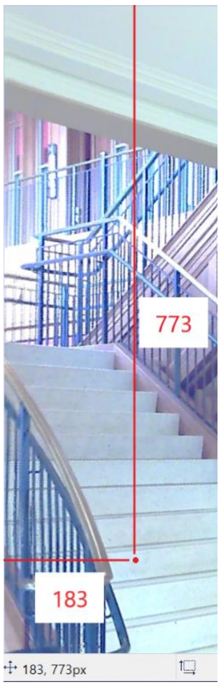
[1] Garcia, Rafael, "A proposal to estimate the motion of an underwater vehicle through visual mosaicking", 2002. pp. 133-134. https://www.researchgate.net/publication/265623150_A_proposal_to_estimate_the_motion_of_an_underwater_vehicle_through_visual_mosaicking

[2] R. Hartley and A. Zisserman, "Multiple View Geometry in Computer Vision," Cambridge University Press, 2000.

Part 2 Mosaicing

`p_in, p_ref = set_cor_mosaic()`

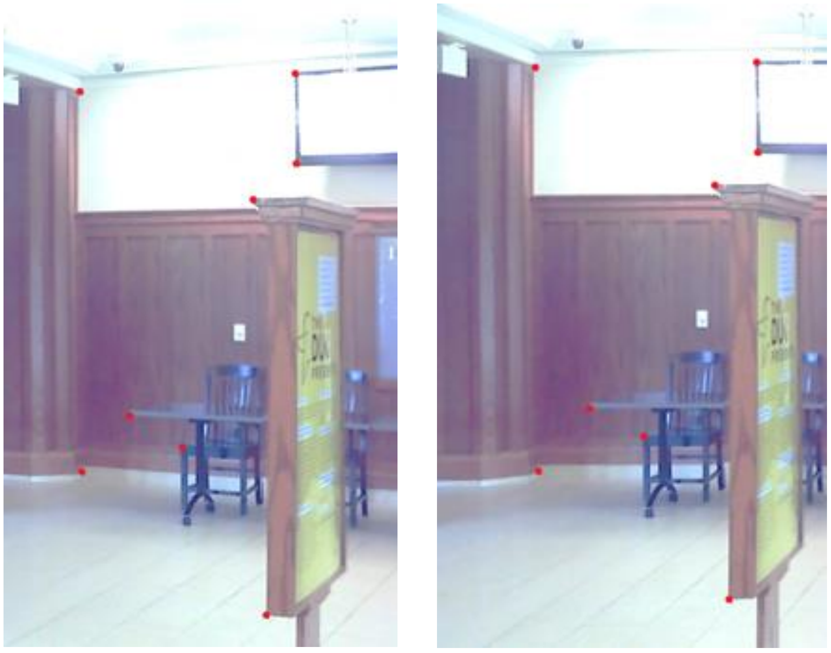
To retrieve coordinates of pixels, I use Windows Paint. One interesting point to note is that Paint gives image coordinates in a similar way matrix indices are done. For example, in the right image, the image coordinate (183, 773) is equivalent to the numpy index (773, 183). This finding of the coordinate mechanism gains importance as several numpy operations are performed.



Criterion for selecting correspondences

I selected correspondences according to the following list of criteria:

- ✓ Is located at the overlapping region of the two images
- ✓ Is a corner point of an object, which can be best distinguished from gradients of the surrounding area
- ✓ has an image intensity that sufficiently differs from the surrounding area



Number of correspondences used

I used **9 correspondences**, which is more than the minimum number of correspondences required (4). The correspondences include points of the television, yellow sign near to the camera, pillars of the entrance, table, chair, and some air conditioning vent on the ceiling.

2018-17119 박수현

`igs_warp, igs_merge = warp_image(igs_in, igs_ref, H)`

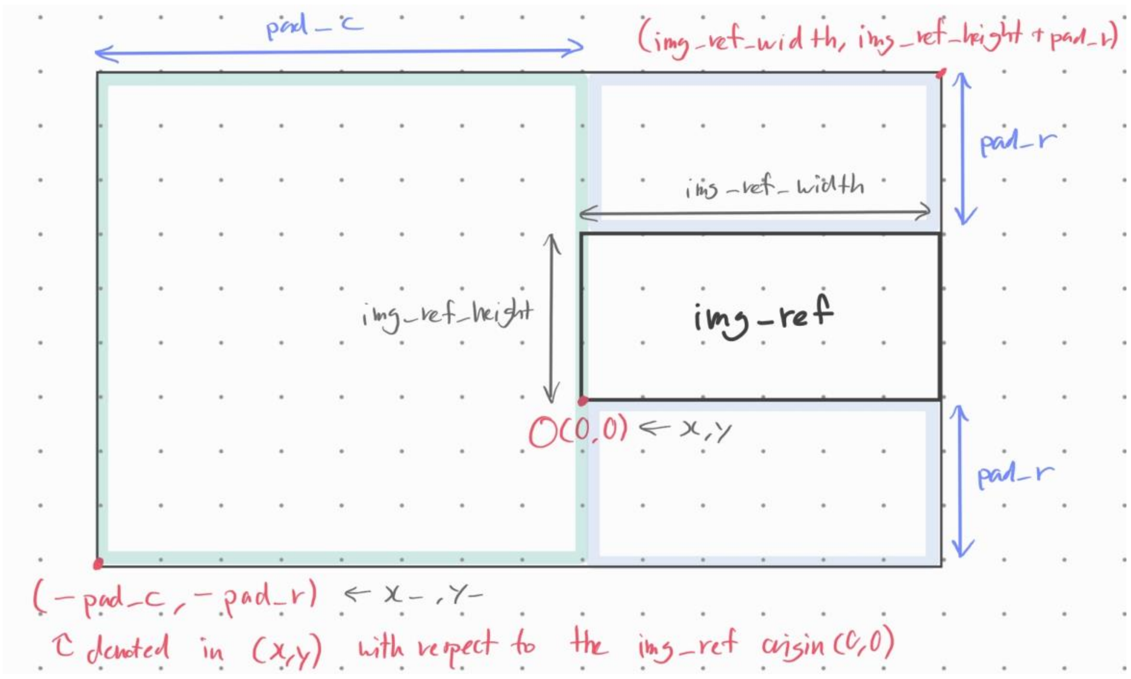
Determining the size of igs_merge and defining the coordinate system

The figure below shows how I formed the igs_merge canvas and denoted the origin of the image.

Igs_merge is formed through zero padding with the padding size of ((pad_r, pad_r), (pad_c, 0), (0, 0)). The optimal igs_merge size that makes both igs_warp and igs_ref best fit to the borders was determined after some calibration attempts. The resulting padding size is calculated from pad_r, pad_c = 461, 1640.

Since the inverse warping method requires pixels from coordinates centered to img_ref, I design the algorithm to operate along the origin (0,0) set at the bottom left corner of img_ref. That is, from the perspective of img_ref, the coordinates of igs_merge's bottom left corner A should be A(-pad_c, -pad_r) and the upper right corner B should B(img_ref_width, img_ref_height + pad_c). Therefore, I looped through coordinates (x_, y_) of igs_merge from A to B and calculated their corresponding locations (x, y) in igs_in. Note that in numpy notations, each locations refer to igs_merge[y_, x_] and igs_merge[y, x], respectively.

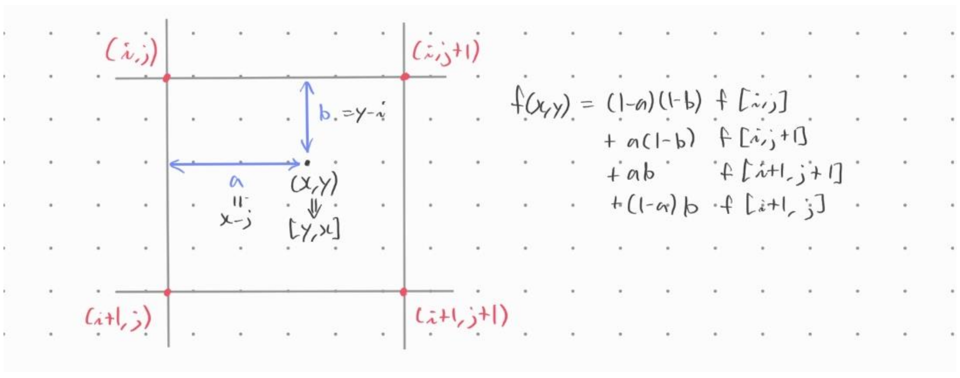
(x, y) is obtained by matrix multiplication of H^{-} and individual homogeneous coordinates of (x_, y_).



pad_c
(img_ref_width, img_ref_height + pad_r)
pad_r
img_ref_width
img_ref_height
img_ref
O(0,0) ← x, y
pad_r
(-pad_c, -pad_r) ← x_, y_
↳ denoted in (x,y) with respect to the img_ref origin (0,0)

Bilinear interpolation

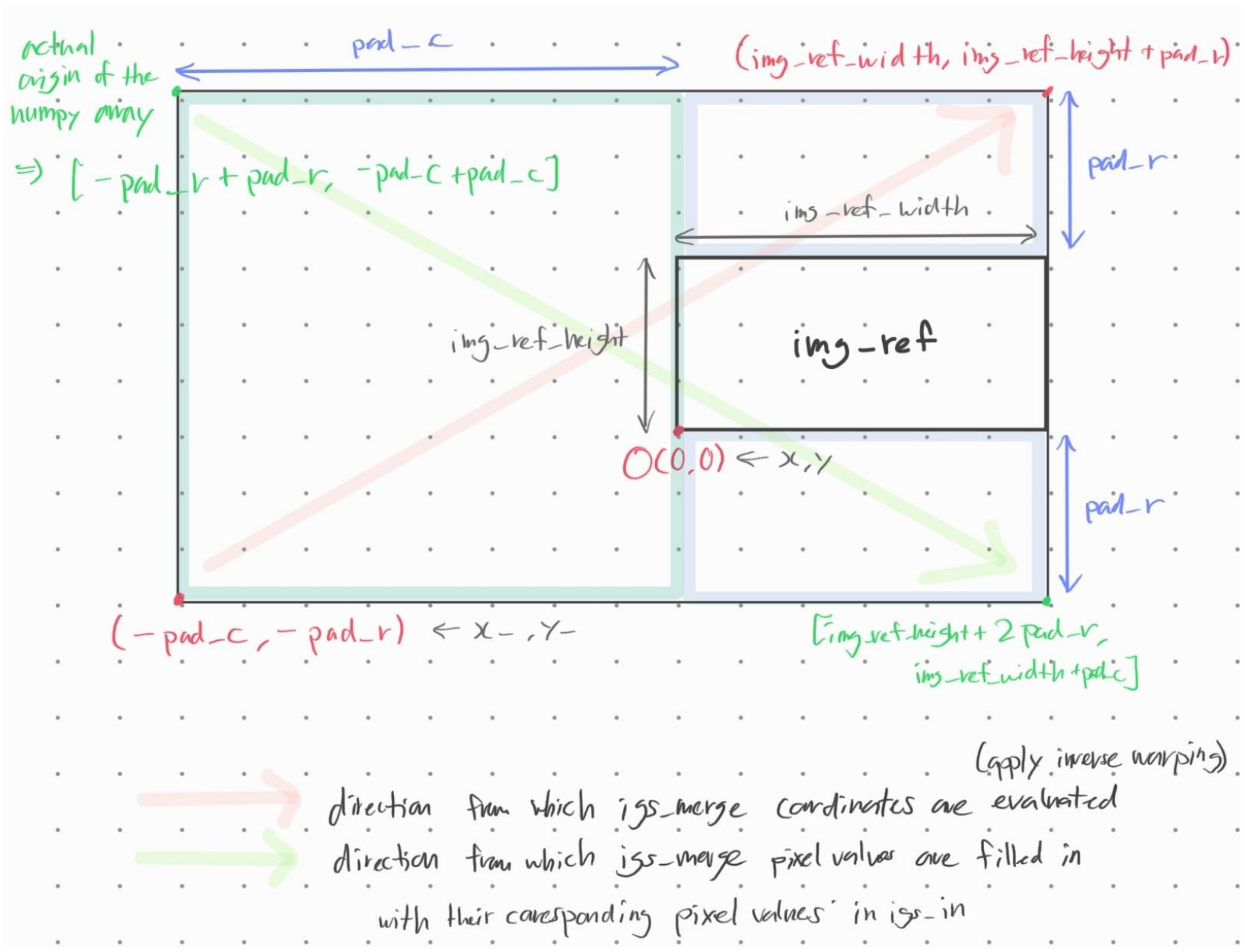
During inverse warping, it is critical to handle (x, y) coordinates that are float values instead of integers. I use bilinear interpolation to interpolate color value from its neighbors. For more convenient numpy operations, instead of using image-wise coordinates of (x, y), I use notations of (y, x). Accordingly, I put out the appropriate neighboring coordinates and distance measures as follows:



2018-17119 박수현

Coordinate selection and pixel value insertion for igs_warp and igs_merge

Numpy operations set the starting point (0,0) differently, namely the upper left corner of igs_merge. While igs_merge coordinates (x_, y_) are looped through from bottom left to upper right (see the red text and arrow in the figure below), when inserting pixel values into the numpy array igs_merge[y_, x_], the indices must be nonnegative. Thus, I added pad_r and pad_c to y_ and x_ respectively so that igs_merge will be filled out starting from indices [0, 0]. In this sense, pixel values are inserted to igs_merge from the upper left to bottom right (see the green text and arrow).



actual origin of the numpy array
⇒ [-pad_r + pad_r, -pad_c + pad_c]
pad_c
(img_ref_width, img_ref_height + pad_r)
pad_r
img_ref_width
img_ref_height
img_ref
O(0,0) ← x,y
pad_r
(-pad_c, -pad_r) ← x_, y_
[img_ref_height + 2 pad_r, img_ref_width + pad_c]
(apply inverse warping)
direction from which igs_merge cordinates are evaluated
direction from which igs_merge pixel values are filled in with their corresponding pixel values in igs_in

The operations for igs_warp and igs_merge are the same until this last coordinate selection and pixel insertion step. The output igs_warp is essentially igs_in warped to match igs_ref's perspective and dimension. It consists of igs_in's pixel coordinates and RGB values that also exist in img_ref. Recalling that coordinates (y_, x_) in igs_merge are are set with respect to img_ref coordinates, and igs_warp is the same dimension with img_ref, I assign the (interpolated) pixel value to igs_warp[y_, x_]. To indicate correct positions within the boundary, y_ and x_ are subject to the condition of being nonnegative. The following is a code snippet for this part.

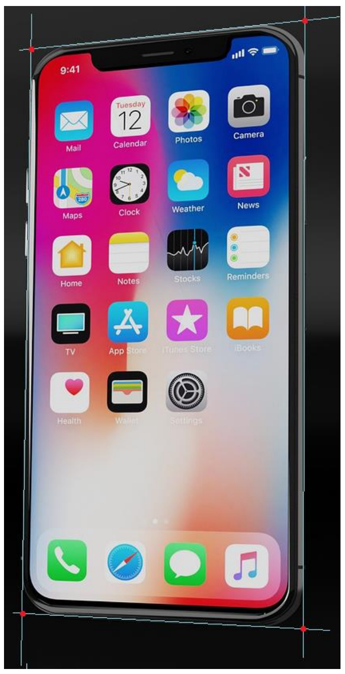
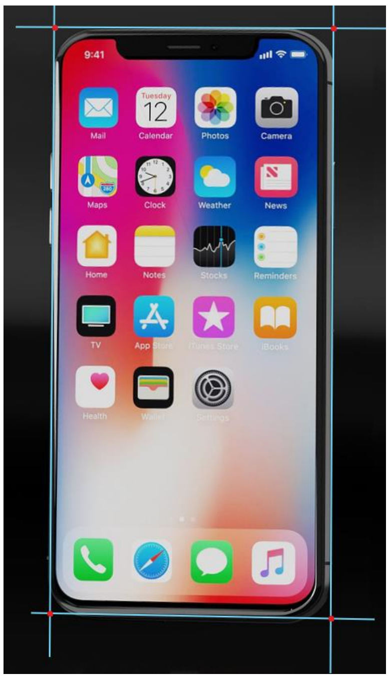
```
if 0 <= y_ < igs_ref_height and 0 <= x_:  # x_ is already smaller than igs_ref_width
    igs_warp[y_, x_] = pixel
igs_merge[y_ + pad_r, x_ + pad_c] = pixel
```

Part 3 Rectification

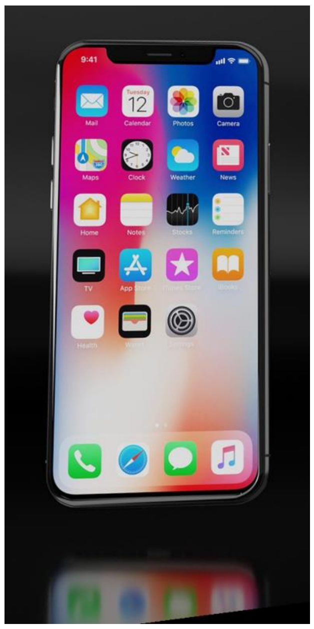
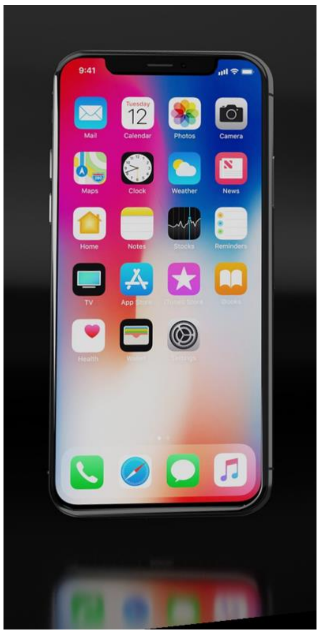
`c_in, c_ref = set_cor_rec()`

For **`c_in`**, I selected the **4 corner points** at which the boundary lines of the Iphone intersect. (left image)

For **`c_ref`**, I first edited the input image by applying the "Free Transform" function in Photoshop, to get the desired output that the **`rectify()`** function should produce. This was an attempt to visually justify the resulting points to form a shape that preserves the original scale and dimension of the input image, rather than naively guessing the positions. Just like **`c_in`**, I retrieved the **4 corner points** that the rectified sample image gives.

I also considered getting the keypoints of the object's reflections in the bottom area, but this results in undesired output (left). This is because the uneven distance between the phone and the reflection affects parallel repositioning of the phone. Although the output images with additional keypoints (left) and without any (right) display a very subtle difference, the right image seems more natural. Therefore, I only choose the 4 keypoints located at the corners of the phone.

`igs_rec = rectify(igs, p1, p2)`

Image rectification basically follows the same procedure done in warp_image(). The difference is that while warp_image() needs to cover two images and a canvas of bigger dimension, rectify() edits only a single image. This means that the coordinate system doesn't have to be readjusted according to igs's origin, and I can directly loop through the numpy array coordinates from the upper left to the lower right. Therefore, igs[y_, x_] is assigned a bilinear-interpolated pixel value from the position of the inversely-warped correspondence (x, y).