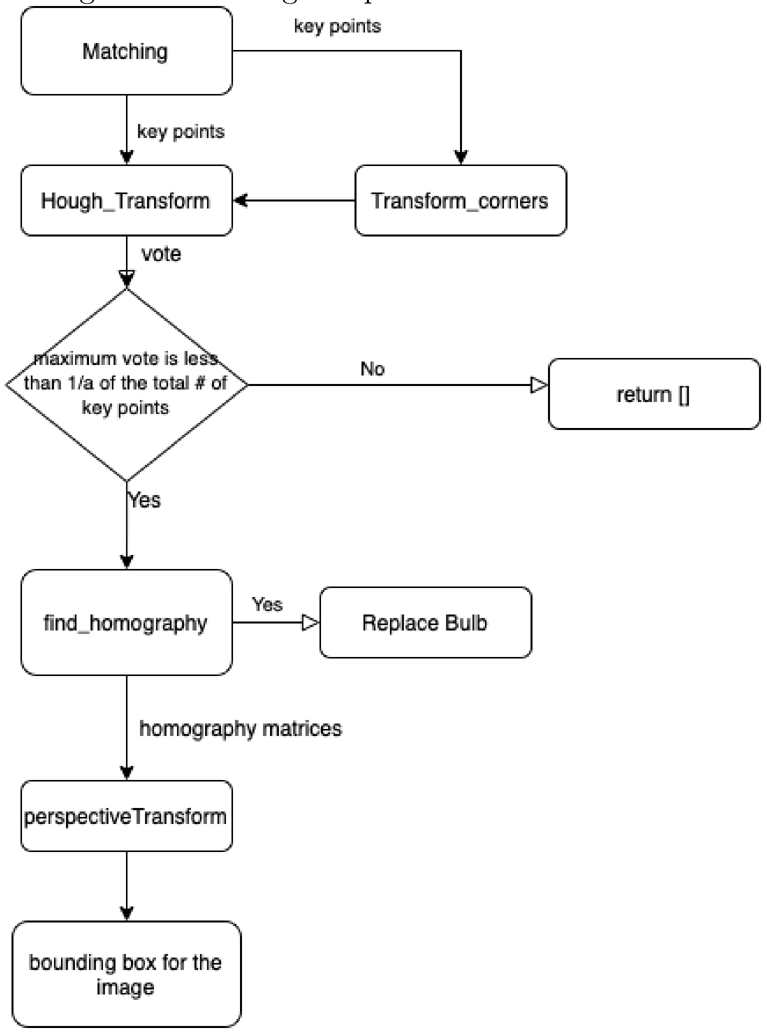
Project **Detection**

Solutions by **Richard Xu, Tianchong Jiang** `richard1xur@uchicago.edu`
and **Tianchong Jiang** `tianchongj@uchicago.edu`

We implemented a multi-instance object detection program in the attached file **detection.ipynb**. The file is attached along with this write-up file. To run, simply run the Jupiter notebook. Once an input and a reference image are provided as input, our program will pass through the following component:



and the result will be the input image along with bounding boxes that indicate the detected object. Now I will briefly go over each component and how they are achieved.

Matching. We used SIFT Brute Force Matcher by applying BFMatcher.knnMatch() to keypoints, which applies ratio test. Then we pass all the matched keypoints to the function transform corners.

Transform Corners. Once we have collected all the keypoints, we would iteratively pass each match, or a pair of reference keypoint and input keypoint, to calculate the corners of the bounding boxes in the input image implied by the input keypoint. First off, we would find the corners of the bounding rectangles in the reference image. Then we would use this to find the bounding rectangles in the input image. Let p denotes the input keypoint and p denotes the reference keypoint. Let s be the scale of the input image and s' be the scale of the reference image. Finally, let θ denote the difference of angle between the input image and the reference image. Let u, v denote the upper right corner coordinates and lower left coordinate of the reference image respectively. Note that the offset $\triangle$ is given by

$$\triangle = \begin{bmatrix} — u^T - p^T — \\ — v^T - p^T — \end{bmatrix}$$

which stacks the two coordinate vector horizontally and gives us a 2×2 matrix that measures the distance between the keypoint and two corner (lower left and upper right) respectively in the reference image. If we translate this offset up to scale and rotated to the same angle as the keypoint in the input image, we would find the coordinates of the lower left and upper right corners in the input image, which is

$$T = \frac{s}{s'} R \triangle^T$$

Finally, we may obtain an equation for the bounding box's corners in the input image by

$$x = \begin{bmatrix} — p^T — \\ — p^T — \end{bmatrix} + T$$

where x is the coordinate of the lower left and upper right corners in the input image and R is the rotation matrix given by

$$R = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$$

This provides the query that supports the lookup operation described in step 2.

Hough Transform and Voting Scheme. In this step, we would like to filter out all the keypoints that is not included in one specific bounding box. These keypoints could be noise or keypoints for another instance. For each pair of keypoints of the match, we apply the transform corners procedure described above. Now we have a list of four-tuple, where each tuple entry indicates the value of one coordinate. We divide each entry into 3 partitions evenly ranging from the lowest value to the highest value, and let each input votes by where its calculated corner belongs to the range. Then we index each input point by its vote. Counting all patterns, we collect the 4-tuple vote pattern that appears above a threshold. For each such pattern of vote index, we trace back all the input keypoints that have this vote index. Then these keypoints form one partition of key points for the bounding box. We return a list of partitions of input keypoints along with their corresponding reference keypoints.

Homography matrices. We calculate the homography matrix between input keypoints and its corresponding reference keypoints for all partitions by using open cv's findHomography with RANSAC. Then we return these homography matrics.

Perspective Transform and the bounding box. We apply these homography matrices to the original reference image's bounding box and apply these bounding boxes by homogrpahy.

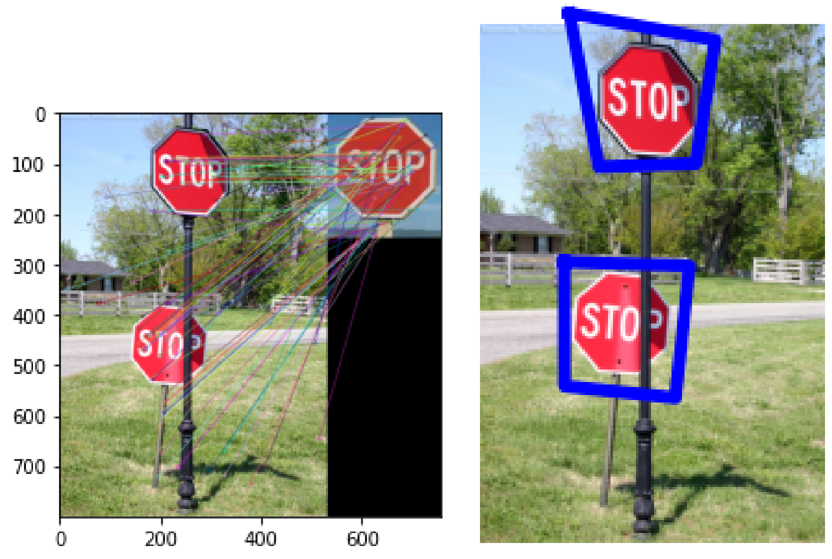
Tuning Hyperparameter. The hyperparameter that particularly matters in the finding of keypoints and SIFT descriptors "contrastThreshold". It is the contrast threshold used to filter out weak features. A large contrast threshold means that the SIFT function will only return the keypoints with high-qualities.

Note that we have a trade off between quality of keypoints and quantity of keypoints. When the contrast threshold is high, there are only a few of keypoints. However, the keypoints has high quality so they are less likely to be mismatch. Then, if the object that we are trying to detect is has a lot of high quality features such as sharp corners and edges, we can set this parameter to be higher to reduce false positives in the matching. When the contrast threshold is high, the SIFT function returns a lot of keypoints but with poor quality. If the object does not have many high quality features, we show set this parameter to be lower so we can have enough keypoints to continue with matching and hough transform.

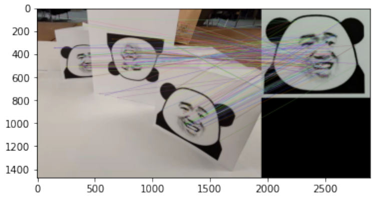
In multiple instance detection, a hyperparameter of particular importance is the threshold of voting i.e. how many votes a partition has to have to be classified as corner. First, we check if the partition with the most vote has votes more than a fraction of total votes. Note that if we are trying to detect many instances, we should set the fraction to be small because each instance will not have a large fraction of votes.

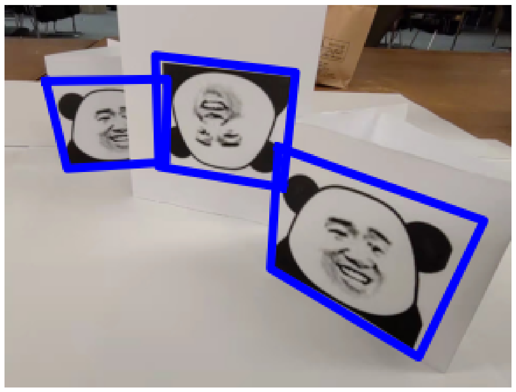
On the other hand, we should not set this fraction too small because even when there is not any instance in the image, some partition might still have many votes because of mismatches. Second, after identifying the partition with the most votes, we check if there is other partitions with votes no less than a fraction of the partition with the most votes. We do so based on the assumption that if there are multiple instance of the same object, these instances are likely to have similar numbers of votes. However, because of difference facing of instances and occlusions, the multiple instances will have different number of votes so we cannot set the fraction too high. We experimented with different values of this fraction and found 1/2 to be a value that works when there is not much noise in the background.

Qualitative evaluation. For single instance objects, our program runs fairly well.
Here are the output of our code:



We have also chosen our own object:





From right to left, the instances are occluded, inverted, and not facing the camera. We can see that our detector deals very well with these three situations.