Homework 3

Shiyu Mou

8708001134

shiyumou@usc.edu

Code Explanation

Here we only explain codes that are written by me.

```
def detect_and_match_feature(self, img1, img2):
    # find keypoint and descriptor
    sift = cv2.xfeatures2d.SIFT_create()
    kp1, des1 = sift.detectAndCompute(img1, None)
    kp2, des2 = sift.detectAndCompute(img2, None)

    # find feature matching
    matcher = cv2.DescriptorMatcher_create(cv2.DescriptorMatcher_FLANNBASED)
    knn_matches = matcher.knnMatch(des1, des2, 2)
    # run ratio test, add points that passed ratio test to list
    ratio_thresh = 0.8
    good_matches = []
    p1 = []
    p2 = []
    for m,n in knn_matches:
      # make sure the the best match is better than
      # the second-best match by at 0.8
      if m.distance < ratio_thresh * n.distance:
        good_matches.append(m)
        p1.append(kp1[m.queryIdx].pt)
        p2.append(kp2[m.trainIdx].pt)
        p1 = np.array(p1).reshape(-1, 1, 2)
        p2 = np.array(p2).reshape(-1, 1, 2)
    return p1, p2, good_matches, kp1, kp2

 def compute_essential(self, p1, p2):
    ## find essential matrix by Ransac, the output is E and a binary mask
    E, mask = cv2.findEssentialMat(p1,p2,self.intrinsic,
                                   method=cv2.RANSAC,prob=0.999,
                                   threshold=1.0)
```

```
    return E, mask
  def compute_pose(self, p1, p2, E):
    # decompose the essenitial matrix to R and T (trans)
    points, R, trans, mask = cv2.recoverPose(E, p1, p2, self.intrinsic)
    return R, trans
  def triangulate(self, p1, p2, R, trans, mask):
    # triangulate 2d matchings to 3d
    # construct extrinsic matrix for camera 1 and camera
    # here we align world coordinates with the first camera
    extrinsic1 = np.eye(3, 4)
    extrinsic2 = np.hstack((R, trans))
    # camMat1 = np.matmul(self.intrinsic, extrinsic1)
    # camMat2 = np.matmul(self.intrinsic, extrinsic2)
    print("camera matrix 1 is: \n", camMat1)
    print("camera matrix 2 is: \n", camMat2)
    # apply mask
    p1 = p1[np.where(mask==1)[0]]
    p2 = p2[np.where(mask==1)[0]]
    # undistort 2d points to the normalized image plane
    undist_p1 =
cv2.undistortPoints(p1,cameraMatrix=self.intrinsic,distCoeffs=None).reshape(-1, 2)
    undist_p2 =
cv2.undistortPoints(p2,cameraMatrix=self.intrinsic,distCoeffs=None).reshape(-1, 2)
    # triagulate normalized 2d points to 3d, notice the results are homogenous
coordinates
    # so the divide the first 3 dimension by the 4th dimension
    point_4d = cv2.triangulatePoints(extrinsic1, extrinsic2, undist_p1.T, undist_p2.T)
    point_3d = point_4d/np.tile(point_4d[-1, :], (4, 1))

    point_3d = point_3d[:3, :].T
    print(point_3d.shape)
    return point_3d
```

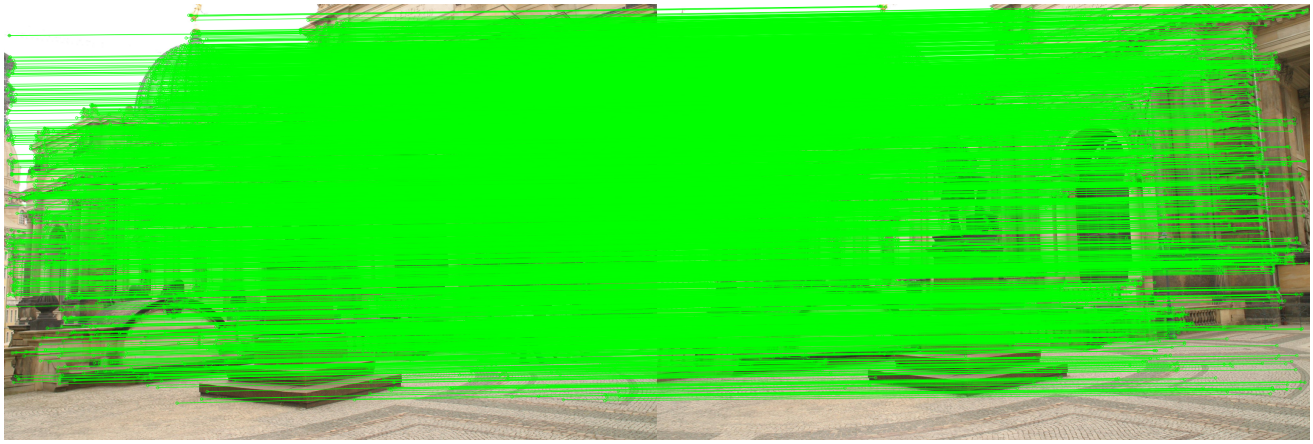
Results

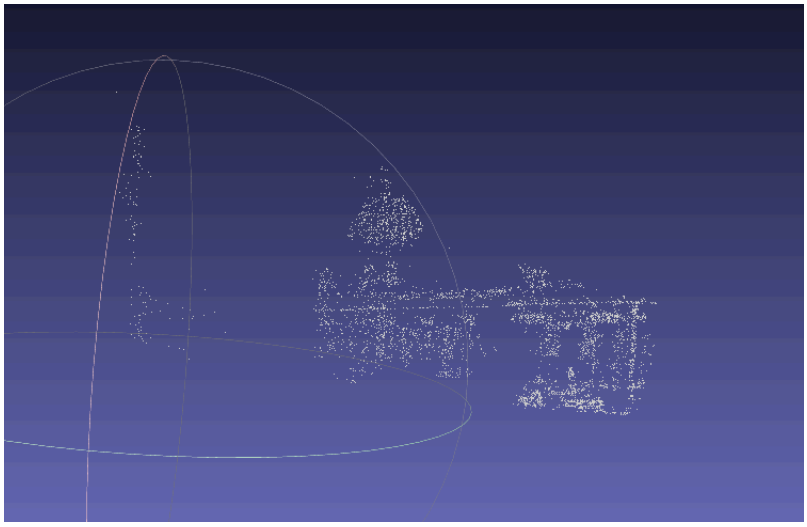
Note that only the matrices of second camera are shown as the world coordinate is aligned with the first camera.

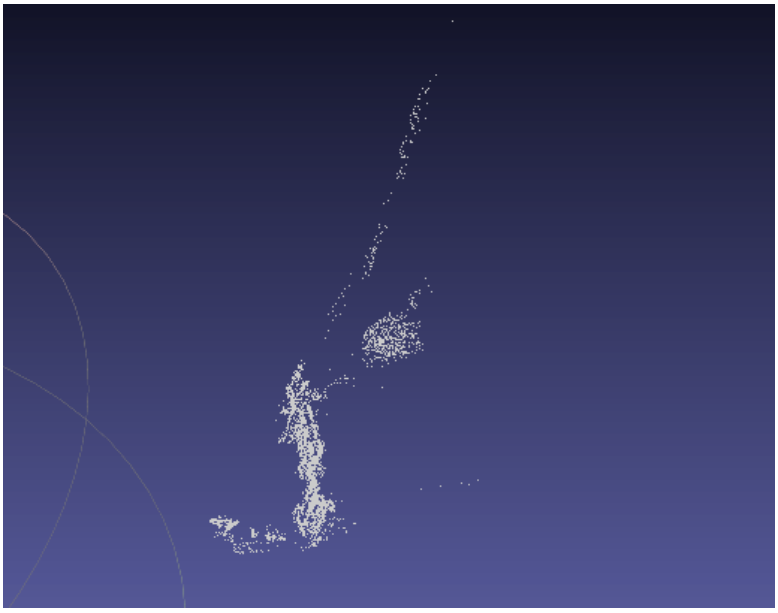
0-1:

```
Test Ratio: 0.9
R:
[[ 9.97739563e-01  2.13911874e-03  6.71653870e-02]
 [-8.59177937e-04  9.99817598e-01 -1.90796599e-02]
 [-6.71939495e-02  1.89788245e-02  9.97559410e-01]]
t:
[[-0.97755596]
 [ 0.0382311 ]
 [ 0.20717803]]
Projection Matrix:
 [[ 1.33522862e+03  1.75801346e+01  8.61112256e+02 -1.19943644e+03]
 [-3.48311258e+01  1.39601853e+03  4.72954065e+02  1.56738317e+02]
 [-6.71939495e-02  1.89788245e-02  9.97559410e-01  2.07178030e-01]]
```

Inner match

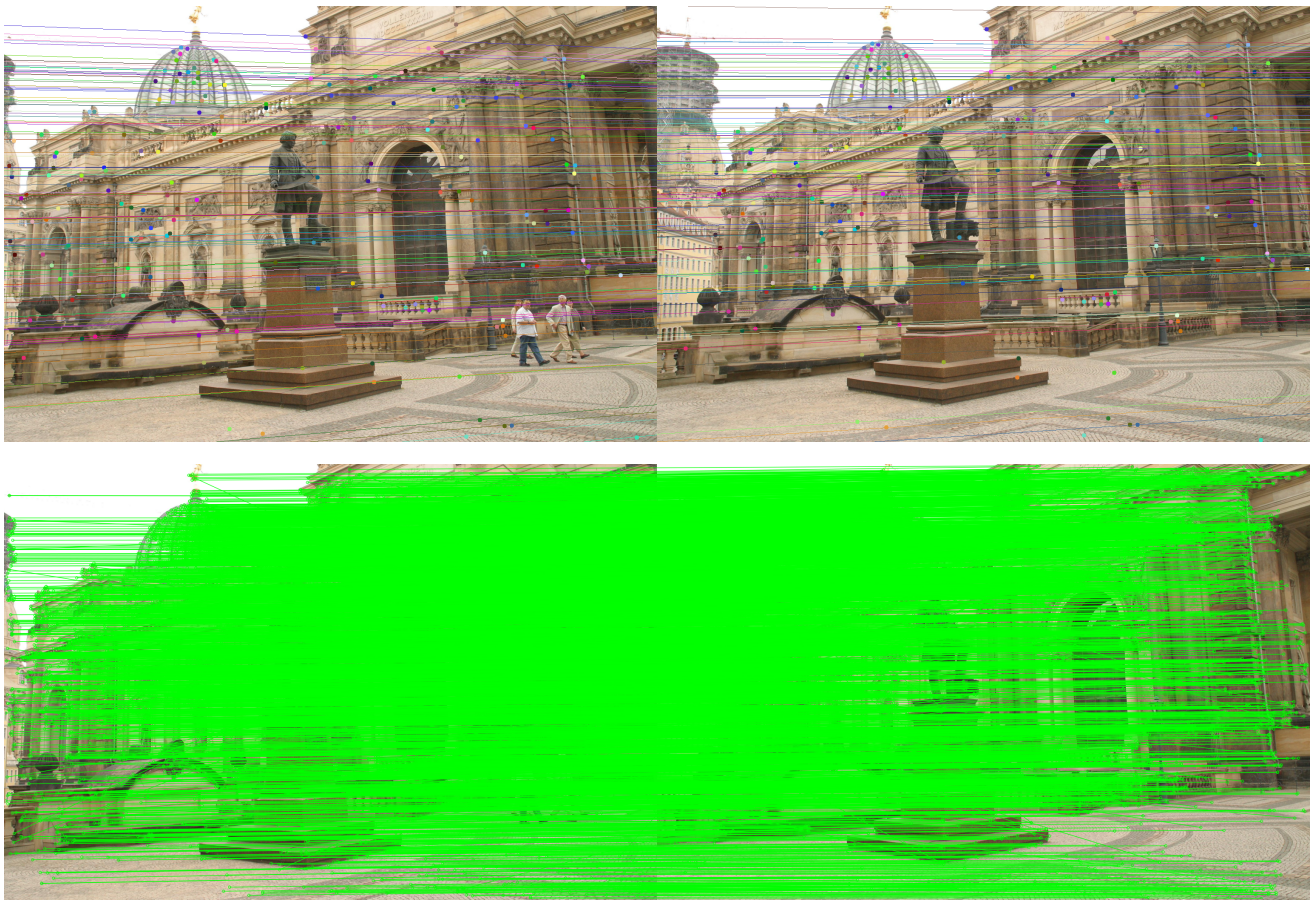




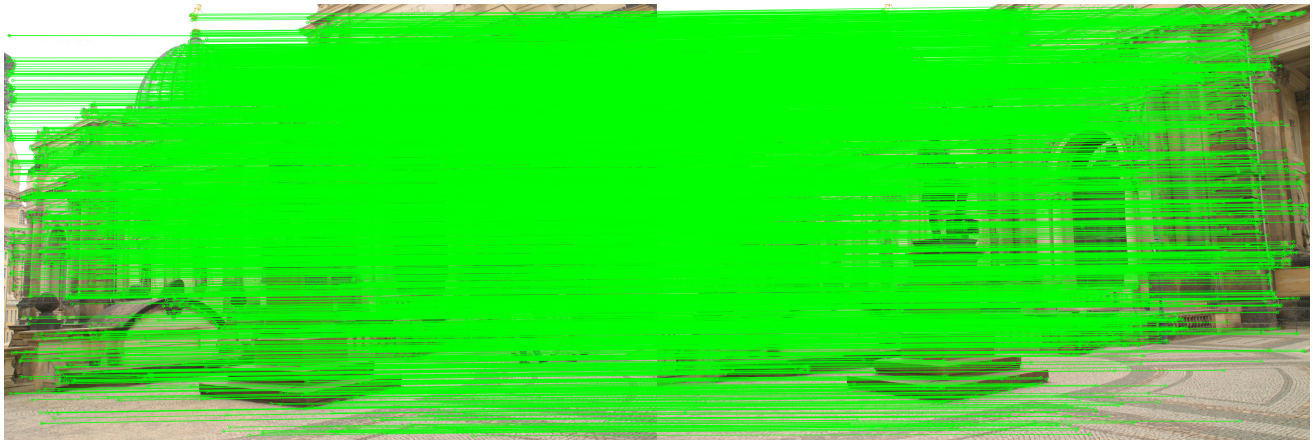


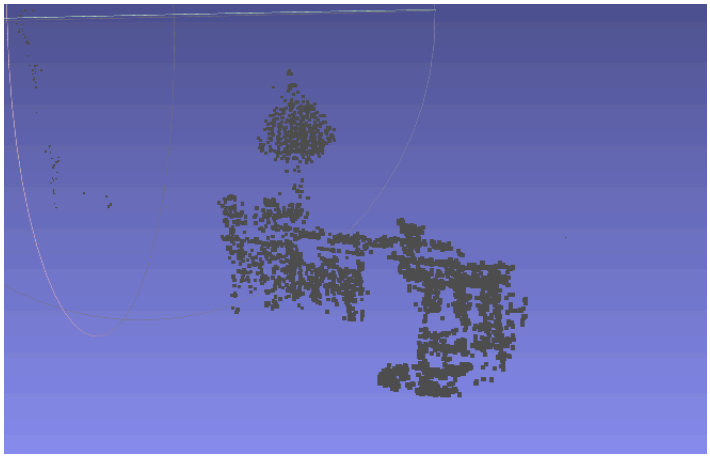
0-2:

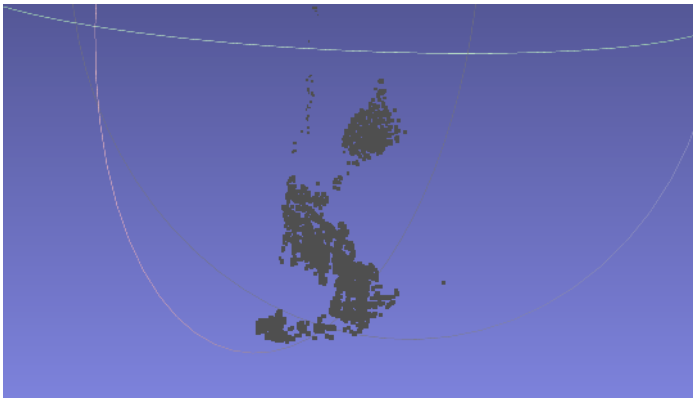
```
Test Ratio: 0.8
R:
[[ 0.9972597   0.00653782  0.07369092]
 [-0.00562026  0.99990417 -0.01265196]
 [-0.07376657  0.01220312  0.99720087]]
t:
[[-0.99890188]
 [ 0.04277542]
 [ 0.01911292]]
Projection Matrix:
 [[ 1.32950311e+03  1.84799536e+01  8.69907360e+02 -1.37384869e+03]
 [-4.47241380e+01  1.39274643e+03  4.81688315e+02  6.88882675e+01]
 [-7.37665745e-02  1.22031233e-02  9.97200871e-01  1.91129238e-02]]
```



Inner

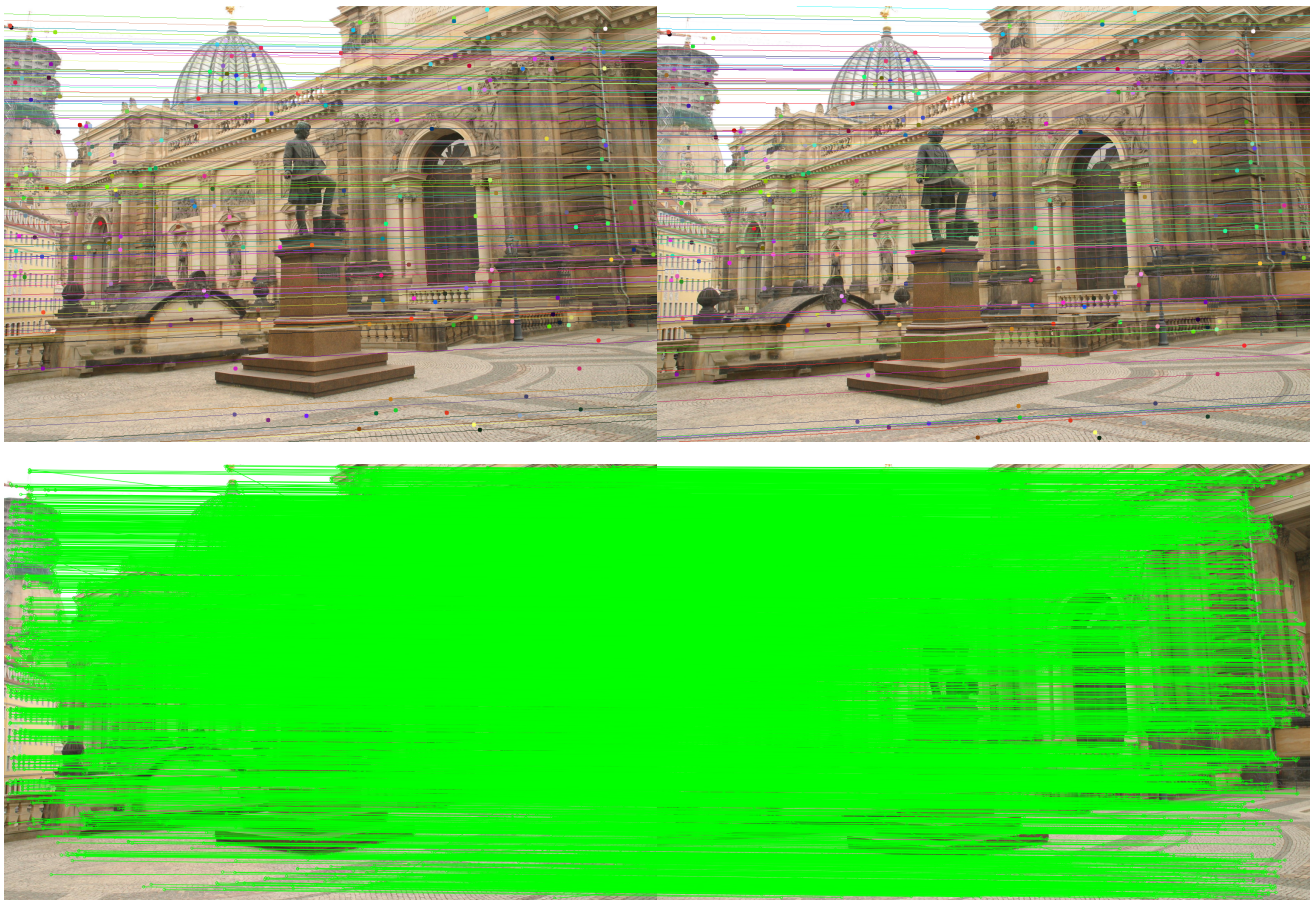




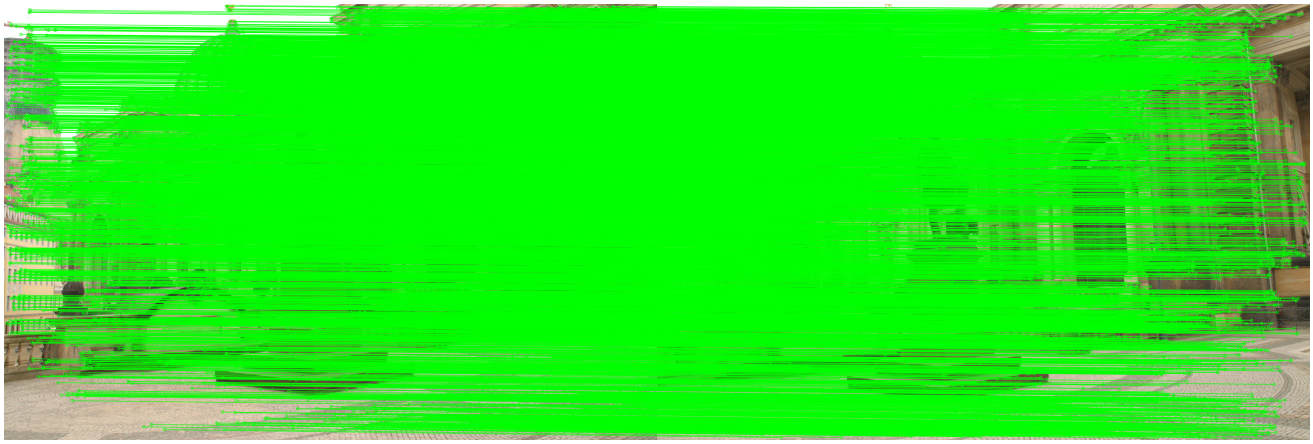


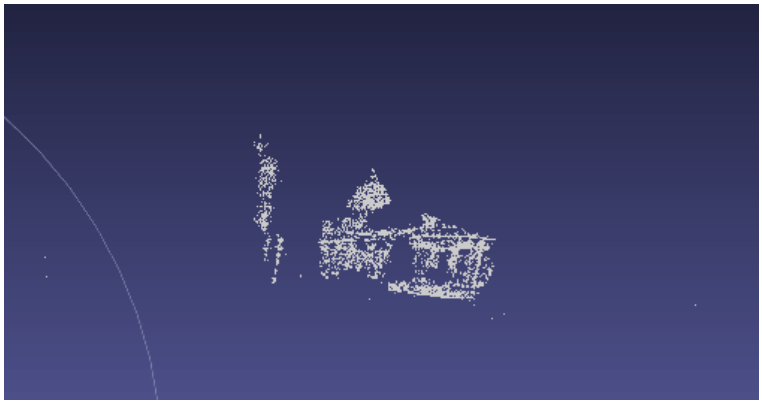
1-2:

```
Test Ratio: 0.7
R:
[[ 0.9998798   0.00384662  0.01501975]
 [-0.00393659  0.99997446  0.00596526]
 [-0.01499642 -0.00602367  0.9998694 ]]
t:
[[-0.99675237]
 [ 0.06185202]
 [-0.05156591]]
Projection Matrix:
 [[ 1.37837625e+03  7.11161200e-01  7.90403224e+02 -1.42525689e+03]
 [-1.29668858e+01  1.38371894e+03  5.08842079e+02  5.99588197e+01]
 [-1.49964248e-02 -6.02367457e-03  9.99869403e-01 -5.15659128e-02]]
```



Inner





Analysis

The method works fine in reconstruction the buildings. From the 3D plot we can discover the 3D structure of buildings, we can even see a lit bit of the sculpture of the horse.

However, this method is considered "indirect method", which relies on feature matching. Only matched features are reconstructed. This results in sparse reconstruction, we can observe holes everywhere.

One way to get denser reconstruction is to use direct method, similar to the stereo method we talked about in class. After we get camera pose, we first rectify the camera, then construct points by matching neighbor color intensities for each pixel and compute disparity map. OF course this method may fail in some area like white wall, sky, floor, due to low color intensity gradient, but still we get much denser results.

Speaking of the accuracy of reconstruction, since we totally have 3 images, we can actually run a global bundle adjustment to optimization R, t and 3D locations.