CSE 559A/Fall 2020. Problem Set 3 Solution Key.

It is a violation of the academic integrity policy to scan, copy, or otherwise share these solutions

1. (a)

$$Pr = \frac{\binom{N-J}{K}}{\binom{N}{K}}, \text{ where } \binom{N}{K} = \frac{N!}{K!(N-K)!}$$

(b) Number of trials T is such that:

$$1 - \left(1 - \frac{\binom{N-J}{K}}{\binom{N}{K}}\right)^T \geq P \Rightarrow T \geq \frac{\log(1-P)}{\log\left(1 - \frac{\binom{N-J}{K}}{\binom{N}{K}}\right)}$$

(c)

$$Pr = \frac{\binom{I_1}{K} + \binom{I_2}{K}}{\binom{N}{K}}$$

2 (a)

```
def fitLine(points, eps, numit=10):

    inlier_idx = list(range(0,points.shape[0]))
    for it in range(numit):
        pts = points[inlier_idx,:]
        p0m = pts[:,0]-np.mean(pts[:,0])
        p1m = pts[:,1]-np.mean(pts[:,1])
        m = np.sum(p0m*p1m) / np.sum(p0m**2)
        b = np.mean(pts[:,1]-pts[:,0]*m)

        L = np.float32([m,b])
        err = (points[:,1]-points[:,0]*m-b)**2
        inlier_idx = np.where(err < eps)[0]

        if len(inlier_idx) < 2:
            break

    return L
```

A single least squares fit works well when there's low-variance noise and no outliers. For outliers, the iterative estimation is better, but this breaks down when the number of outliers is too high because the iterations converge to a poor local minimum.

(b)

```
def ransac(points, K, N, eps):
    best_set = []
    for it in range(N):
        idx = np.random.choice(points.shape[0],K,replace=False)

        pts = points[idx,:]
        p0m = pts[:,0]-np.mean(pts[:,0])
        p1m = pts[:,1]-np.mean(pts[:,1])
        m = np.sum(p0m*p1m) / np.sum(p0m**2)
        b = np.mean(pts[:,1]-pts[:,0]*m)

        err = (points[:,1]-points[:,0]*m-b)**2
        inlier_idx = np.where(err < eps)[0]
```

```
        if len(inlier_idx) > len(best_set):
            best_set = inlier_idx
    pts = points[best_set,:]
    p0m = pts[:,0]-np.mean(pts[:,0])
    p1m = pts[:,1]-np.mean(pts[:,1])
    m = np.sum(p0m*p1m) / np.sum(p0m**2)
    b = np.mean(pts[:,1]-pts[:,0]*m)

    return np.float32([m,b])
```

As expected, a larger number of runs helps. But it is also beneficial to have smaller values of K, because this increases the chance that the drawn samples will all be inliers.

3 (a)

$$x_2 = (x_1 - W/2)\frac{f_2}{f_1} + W/2, \quad y_2 = (y_1 - H/2)\frac{f_2}{f_1} + H/2$$

(b) Consider a world co-ordinate system where the plane is defined by $z = 0$, and let the camera projection matrices for the two cameras, in that co-ordinate system, be P_1 and P_2. Since everything's calibrated, we assume we know P_1 and P_2, but these are general 4×3 matrices.

Let $\tilde{p}_1$ and $\tilde{p}_2$ be the homogeneous 2D co-ordinates representing the projection of a world point with 4D homogeneous co-ordinates p. This point lies on the $z = 0$ plane, and so the third element of p is 0 (note that this is true no matter what the fourth element / scaling factor is). Define p^+ to be a vector made of the first, second, and fourth element of p, and P_1^+, P_2^+ to be 3×3 matrices composed of the first, second, and fourth columns of P_1 and P_2 respectively. Then,

$$\tilde{p}_1 \sim P_1 p \Rightarrow \tilde{p}_1 = \lambda_1 P_1 p = \lambda_1 P_1^+ p^+; \quad \tilde{p}_2 \sim P_2 p \Rightarrow \tilde{p}_2 = \lambda_2 P_2 p = \lambda_2 P_2^+ p^+,$$

for some scalar values λ_1 and λ_2. Then it follows that,

$$\tilde{p}_2 = \frac{\lambda_2}{\lambda_1}\left(P_2^+ (P_1^+)^{-1}\right)\tilde{p}_1 \sim \left(P_2^+ (P_1^+)^{-1}\right)\tilde{p}_1.$$

Hence, all pairs of projected co-ordinates $\tilde{p}_2$ and $\tilde{p}_1$ of points on the plane are related by the homography $P_2^+ (P_1^+)^{-1}$. (Q: When is P_1^+ not invertible ? When the plane is exactly aligned such that it projects to a line on camera 1's sensor plane.)

4 (a)

```
def getH(pts):
    x=pts[:,0].reshape((-1,1)); y=pts[:,1].reshape((-1,1))
    xx=pts[:,2].reshape((-1,1)); yy=pts[:,3].reshape((-1,1))

    z = np.zeros(yy.shape,dtype=np.float32); o = np.ones(yy.shape,dtype=np.float32)

    r1 = [z, z, z, -x, -y, -o, yy*x, yy*y,yy]
    r2 = [x, y, o, z,z,z, -xx*x, -xx*y, -xx]
    r3 = [-yy*x,-yy*y,-yy, xx*x, xx*y, xx ,z,z,z]

    A = np.concatenate([ np.concatenate(r1,axis=1),
                         np.concatenate(r2,axis=1),
                         np.concatenate(r3,axis=1)],
                       axis=0)
    u,s,v = np.linalg.svd(A,full_matrices=True)
    H = v[-1,:].reshape((3,3))

    return H
```

(b)

```python
def splice(src,dest,dpts):
    ht = src.shape[0]; wt = src.shape[1]
    spts = np.float32([[0,0],[wt-1,0],[0,ht-1],[wt-1,ht-1]])

    H = getH(np.concatenate([dpts,spts],axis=1))

    dpts = np.int64(dpts)
    x = np.float32(range(np.min(dpts[:,0]),np.max(dpts[:,0])+1))
    y = np.float32(range(np.min(dpts[:,1]),np.max(dpts[:,1])+1))
    x,y = np.meshgrid(np.float32(x),np.float32(y))
    x = np.reshape(x,[-1,1]); y = np.reshape(y,[-1,1])

    xyd = np.concatenate([x,y],axis=1)
    xydH = np.concatenate([xyd,np.ones((x.shape[0],1))],axis=1)
    xysH = np.matmul(H,xydH.T).T; xys = xysH[:,0:2] / xysH[:,2:3]

    cnd = np.logical_and(xys[:,0] > 0,xys[:,1] > 0)
    cnd = np.logical_and(cnd,xys[:,0] < wt-1); cnd = np.logical_and(cnd,xys[:,1] < ht-1)
    idx = np.where(cnd)[0]; xyd = np.int64(xyd[idx,:]); xys = xys[idx,:]

    # Bilinear interpolation
    xysf = np.int64(np.floor(xys)); xysc = np.int64(np.ceil(xys))
    xalph = xys[:,0:1] - np.floor(xys[:,0:1]); yalph = xys[:,1:2] - np.floor(xys[:,1:2])
    xff = src[xysf[:,1],xysf[:,0],:]; xfc = src[xysf[:,1],xysc[:,0],:]
    xcf = src[xysc[:,1],xysf[:,0],:]; xcc = src[xysc[:,1],xysc[:,0],:]

    comb = dest.copy()
    comb[xyd[:,1],xyd[:,0],:] = (1-xalph)*((1-yalph)*xff+yalph*xcf) + xalph*((1-yalph)*xfc+yalph*xcc)
    return comb
```

5 (a)

```python
def census(img):
    W = img.shape[1]; H = img.shape[0]
    c = np.zeros([H,W],dtype=np.uint32)

    inc = np.uint32(1)
    for dx in range(-2,3):
        for dy in range(-2,3):

            if dx == 0 and dy == 0:
                continue

            cx0 = np.maximum(0,-dx); dx0 = np.maximum(0,dx)
            cx1 = W-dx0; dx1 = W-cx0
            cy0 = np.maximum(0,-dy); dy0 = np.maximum(0,dy)
            cy1 = H-dy0; dy1 = H-cy0

            c[cy0:cy1,cx0:cx1] = c[cy0:cy1,cx0:cx1] + \
                    inc*( img[cy0:cy1,cx0:cx1] > img[dy0:dy1,dx0:dx1])
            inc = inc*2
    return c
```

(b)

```
def smatch(left,right,dmax):
    lc = census(left); rc = census(right)
    d = np.zeros(lc.shape); best = hamdist(lc,rc)
    W = lc.shape[1]
    for i in range(1,dmax+1):
        sc = hamdist(lc[:,i:],rc[:,0:(W-i)])
        yx = np.where(sc < best[:,i:])
        best[yx[0],yx[1]+i] = sc[yx[0],yx[1]]
        d[yx[0],yx[1]+i] = i
    return d
```