CS231A

Computer Vision: From 3D Reconstruction to Recognition

Luke Jaffe

Problem Set 1

Due 01/26/2018

1 Projective Geometry Problems

d) You have explored whether these three properties hold for affine transformations. Do these properties hold under any projective transformation? Justify briefly in one or two sentences (no proof needed).

No, for instance, parallel lines in the world reference system are not still parallel after applying radial distortion, which can be encoded as part of a projective transformation.

2 Affine Camera Calibration

a) You will construct a linear system of equations and solve for the camera parameters to minimize the least-squares error. After doing so, you will return the 3×4 affine camera matrix composed of these computed camera parameters. In your written report, submit your code as well as the camera matrix that you compute.

```
def compute_camera_matrix(real_XY, front_image, back_image):
    '''
    COMPUTE_CAMERA_MATRIX
    Arguments:
         real_XY - Each row corresponds to an actual point on the 2D plane
         front_image - Each row is the pixel location in the front image where Z=0
         back_image - Each row is the pixel location in the back image where Z=150
    Returns:
        camera_matrix - The calibrated camera matrix (3x4 matrix)
    '''
    # Create real point matrix
    front_z = np.zeros(front_image.shape[0])[:, np.newaxis]
    back_z = 150*np.ones(front_image.shape[0])[:, np.newaxis]
    both_z = np.concatenate([front_z, back_z], axis=0)
    twice_real_XY = np.concatenate([real_XY, real_XY], axis=0)
    real_ones = np.ones(twice_real_XY.shape[0])[:, np.newaxis]
    A = np.concatenate([twice_real_XY, both_z, real_ones], axis=1)

    # Create projected point matrix
    both_image = np.concatenate([front_image, back_image], axis=0)
    image_ones = np.ones(both_image.shape[0])[:, np.newaxis]
    b = np.concatenate([both_image, image_ones], axis=1)

    # Solve with least-squares
    xt, _, _, _ = np.linalg.lstsq(A, b)
    camera_matrix = xt.T

    return camera_matrix
```

Camera Matrix:

```
[[ 5.31276507e-01  -1.80886074e-02   1.20509667e-01   1.29720641e+02]
 [  4.84975447e-02   5.36366401e-01  -1.02675222e-01   4.43879607e+01]
 [ -2.02336860e-18   5.20417043e-18   3.25260652e-18   1.00000000e+00]]
```

b) After finding the calibrated camera matrix, you will compute the RMS error between the given N image corner coordinates and N corresponding calculated corner locations in rms_error(). Please submit your code and the RMS error for the camera matrix that you found in part (a).

```
def rms_error(camera_matrix, real_XY, front_image, back_image):
    '''
    RMS_ERROR
    Arguments:
         camera_matrix - The camera matrix of the calibrated camera
         real_XY - Each row corresponds to an actual point on the 2D plane
         front_image - Each row is the pixel location in the front image where Z=0
         back_image - Each row is the pixel location in the back image where Z=150
    Returns:
        rms_error - The root mean square error of reprojecting the points back
                    into the images
    '''
    # Create real point matrix
    front_z = np.zeros(front_image.shape[0])[:, np.newaxis]
    back_z = 150*np.ones(front_image.shape[0])[:, np.newaxis]
    both_z = np.concatenate([front_z, back_z], axis=0)
    twice_real_XY = np.concatenate([real_XY, real_XY], axis=0)
    real_ones = np.ones(twice_real_XY.shape[0])[:, np.newaxis]
    A = np.concatenate([twice_real_XY, both_z, real_ones], axis=1)

    # Create projected point matrix
    real_proj = np.concatenate([front_image, back_image], axis=0)

    # Calculate corner locations
    calc_b = np.dot(A, camera_matrix.T)
    calc_proj = calc_b[:, :2]

    # Compute RMS error
    rms_error = np.sqrt(np.sum((calc_proj - real_proj)**2)/real_proj.shape[0])

    return rms_error
```

RMS Error: 0.99383048328

c) Could you calibrate the matrix with only one checkerboard image? Explain briefly in one or two sentences.

No, if all points used for calibration lie on the same plane i.e. one checkerboard image in this case, the point configuration will be degenerate, and the system cannot be solved.

3 Single View Geometry

a) In Figure 2, we have identified a set of pixels to compute vanishing points in each image. Please complete compute_vanishing_point(), which takes in these two pairs of points on parallel lines to find the vanishing point. You can assume that the camera has zero skew and square pixels, with no distortion.

```
def compute_vanishing_point(points):
    '''
    COMPUTE_VANISHING_POINTS
    Arguments:
        points - a list of all the points where each row is (x, y). Generally,
                it will contain four points: two for each parallel line.
                You can use any convention you'd like, but our solution uses the
                first two rows as points on the same line and the last
                two rows as points on the same line.
    Returns:
        vanishing_point - the pixel location of the vanishing point
    '''
    # Unpack points
    (k1x, k1y), (k2x, k2y), (l1x, l1y), (l2x, l2y) = points

    # Compute slopes
    mk = (k2y - k1y) / (k2x - k1x)
    ml = (l2y - l1y) / (l2x - l1x)

    # Compute intercepts
    bk = k1y - (mk * k1x)
    bl = l1y - (ml * l1x)

    # Compute intersection x
    vx = (bl - bk) / (mk - ml)

    # Compute intersection y
    vy = mk * vx + bk

    return vx, vy
```

b) Using three vanishing points, we can compute the intrinsic camera matrix used to take the image. Do so in compute_K_from_vanishing_points().

```
def compute_K_from_vanishing_points(vanishing_points):
    '''
    COMPUTE_K_FROM_VANISHING_POINTS
    Arguments:
        vanishing_points - a list of vanishing points

    Returns:
        K - the intrinsic camera matrix (3x3 matrix)
    '''
    ### Compute w using SVD
    vp = vanishing_points
    # Construct system of equations using vanishing points
    A = np.array([
        [vp[0][0]*vp[1][0]+vp[0][1]*vp[1][1], vp[0][0]+vp[1][0], vp[0][1]+vp[1][1],
1],
        [vp[0][0]*vp[2][0]+vp[0][1]*vp[2][1], vp[0][0]+vp[2][0], vp[0][1]+vp[2][1],
1],
        [vp[1][0]*vp[2][0]+vp[1][1]*vp[2][1], vp[1][0]+vp[2][0], vp[1][1]+vp[2][1],
1],
    ])
    # Perform SVD on system of equations
    u, s, vt = np.linalg.svd(A, full_matrices=True)
    # Solution w to Aw = 0 is last row of matrix V
    w = vt.T[:, -1]

    # Test w
    print('\nTest SVD:')
    null = np.dot(A, w)
    print('null:', null)

    # Construct omega (W) using the elements of w
    W = np.array([
        [w[0], 0,    w[1]],
        [0,    w[0], w[2]],
        [w[1], w[2], w[3]]
    ])

    # Test omega
    print('\nTest omega:')
    vp1 = np.array(vp[0]+(1,))[:, np.newaxis]
    vp2 = np.array(vp[1]+(1,))[:, np.newaxis]
    vp3 = np.array(vp[2]+(1,))[:, np.newaxis]
    print('null1:', np.dot(np.dot(vp1.T, W), vp2))
    print('null2:', np.dot(np.dot(vp1.T, W), vp3))
    print('null3:', np.dot(np.dot(vp2.T, W), vp3))

    # Compute K inverse from omega using cholesky factorization
    C = np.linalg.cholesky(W)
    # Take the (pseudo-)inverse to get K
    K = np.linalg.pinv(C.T)
    # Normalize K
    K /= K[-1, -1]

    return K
```

Results printed as type=int:
Intrinsic Matrix:
```
 [[2594    0  773]
 [   0 2594  979]
 [   0    0    1]]
```

Actual Matrix:
```
 [[2448    0 1253]
 [   0 2438  986]
 [   0    0    1]]
```

c) Is it possible to compute the camera intrinsic matrix for any set of vanishing points? Similarly, is three vanishing points the minimum required to compute the intrinsic camera matrix? Justify your answer.

It is not possible to compute the camera intrinsic matrix for any set of vanishing points; the set of points must create at least 5 constraints to solve for the 5 degrees of freedom. These constraints are based on conditions including assumption of zero skew, assumption of square pixels, having vanishing points corresponding to orthogonal lines, among others.

It is possible to achieve the 5 needed constraints with only two vanishing points. If the two points correspond to orthogonal lines (1 constraint), and there are internal constraints of having zero skew and square pixels (2 constraints), and the metric plane is imaged with known homography (2 constraints), then all 5 constraints are established, and the system can be solved.

d) The method used to obtain vanishing points is approximate and prone to noise. Discuss approaches to refine this process.

To increase precision of vanishing points, multiple collinear points could be collected for each line in each pair of parallel lines. A line of best fit with least-squares error could be computed for collinear points. Additionally, the same vanishing point could be computed multiple times for a set of parallel lines with more than 2 lines e.g., 3+ parallel lines from a checkerboard or floor tiles. These vanishing points could be then be averaged.

e) Identify a sufficient set of vanishing lines on the ground plane and the plane on which the letter A exists, written on the side of the cardboard box, (plane-A). Use these vanishing lines to verify numerically that the ground plane is orthogonal to the plane-A. Fill out the method compute_angle_between_planes() and submit your code and the computed angle.

```
def compute_angle_between_planes(vanishing_pair1, vanishing_pair2, K):
    '''
    COMPUTE_K_FROM_VANISHING_POINTS
    Arguments:
        vanishing_pair1 - a list of a pair of vanishing points computed from lines
within the same plane
        vanishing_pair2 - a list of another pair of vanishing points from a
different plane than vanishing_pair1
        K - the camera matrix used to take both images

    Returns:
        angle - the angle in degrees between the planes which the vanishing point
pair comes from2
    '''
    # Compute omega inverse using camera matrix
    W_inv = np.dot(K, K.T)

    # Unpack vanishing points and add 1 to end
    vanishing_pair1[0] += (1,)
    vanishing_pair1[1] += (1,)
    vanishing_pair2[0] += (1,)
    vanishing_pair2[1] += (1,)
    v1, v2 = np.array(vanishing_pair1)
    v3, v4 = np.array(vanishing_pair2)

    # Compute l1 and l2
    l1 = np.cross(v1, v2)
    l2 = np.cross(v3, v4)

    # Compute cosine of angle using omega
    cos_theta = l1.T.dot(W_inv.dot(l2))/
(np.sqrt(l1.T.dot(W_inv.dot(l1)))*np.sqrt(l2.T.dot(W_inv.dot(l2))))
    theta = np.arccos(cos_theta)

    # Convert from radians to degrees
    theta_deg = np.degrees(theta)

    return theta_deg
```

Result printed in degrees:
Angle between floor and box: 90.027361241

f) Use vanishing points to estimate the rotation matrix between when the camera took Image 1 and Image 2. Fill out the method compute_rotation_matrix_between_cameras() and submit your code and your results.

```
def compute_rotation_matrix_between_cameras(vanishing_points1, vanishing_points2,
K):
    '''
    COMPUTE_K_FROM_VANISHING_POINTS
    Arguments:
        vanishing_points1 - a list of vanishing points in image 1
        vanishing_points2 - a list of vanishing points in image 2
        K - the camera matrix used to take both images

    Returns:
        R - the rotation matrix between camera 1 and camera 2
    '''
    # Unpack vanishing points
    pa1, pa2, pa3 = vanishing_points1[:, :, np.newaxis]
    pb1, pb2, pb3 = vanishing_points2[:, :, np.newaxis]

    # Group vanishing points by image
    va = np.concatenate([pa1, pa2, pa3], axis=1).T
    vb = np.concatenate([pb1, pb2, pb3], axis=1).T

    # Add ones to vanishing points
    one_row = np.ones((va.shape[0], 1))
    va = np.concatenate([va, one_row], axis=1)
    vb = np.concatenate([vb, one_row], axis=1)

    # Compute directions of vanishing points for each image
    K_inv = np.linalg.pinv(K)
    dau = K_inv.dot(va.T)
    dbu = K_inv.dot(vb.T)

    # Normalize to unit vectors
    da = dau/np.linalg.norm(dau, axis=0)
    db = dbu/np.linalg.norm(dbu, axis=0)

    # Solve for rotation
    da_inv = np.linalg.pinv(da)
    R = np.dot(db, da_inv)

    # Check
    print('\nCheck R:')Angle between floor and box: 90.027361241
    a1, a2, a3 = da.T
    b1, b2, b3 = db.T
    print(b1, np.dot(R, a1))
    print(b2, np.dot(R, a2))
    print(b3, np.dot(R, a3))

    return R
```

Rotation between two cameras:
[[0.96154157 0.04924778 -0.15783349]
[-0.01044314 1.00703585 0.04571333]
[0.18940319 -0.06891607 1.00470583]]

Angle around z-axis (pointing out of camera): -2.931986 degrees
Angle around y-axis (pointing vertically): -8.918793 degrees
Angle around x-axis (pointing horizontally): -2.605117 degrees

Problem 1 a) Prove that parallel lines in the world reference system are still parallel in the camera reference system.

We will use k and l, as in the notes, to refer to two parallel lines, with points k_1, k_2 on k and l_1, l_2 on l.

Since k and l are parallel, by definition: $f \cdot (k_1 - k_2) = \cdot (l_1 - l_2)$, $f \in \mathbb{R}$

Let R be a 3x3 rotation matrix, and T be a 3x1 translation vector.

By definition of parallel lines: $(k_1 - k_2) \times (l_1 - l_2) = 0$

We want to show that the rotated and translated lines still have cross product = 0:

$((R \cdot k_1 + T) - (R \cdot k_2 + T)) \times ((R \cdot l_1 + T) - (R \cdot l_2 + T))$

$= (R \cdot k_1 - R \cdot k_2) \times (R \cdot l_1 - R \cdot l_2)$

$= R(k_1 - k_2) \times R \cdot (l_1 - l_2)$ → distributive property

$= R \cdot (k_1 - k_2) \times f \cdot R \cdot (k_1 - k_2)$ → $f \cdot (k_1 - k_2) = (l_1 - l_2)$, $f \in \mathbb{R}$

$= f \cdot (R \cdot (k_1 - k_2) \times R \cdot (k_1 - k_2))$ → compatible w/ scalar multiplication

$= f \cdot 0$ → self cross product of vector is 0 vector

$= 0$ Q.E.D. parallelism is preserved

Problem 1 b) Consider a unit square pqrs in the world reference system where p, q, r, and s are points. Will the same square in the camera reference system always have unit area? Prove or provide a counterexample.

Given a square pqrs. Area = $\|(q-p) \times (s-p)\|$. in this case Area = 1

We can describe the transformation matrix $\begin{bmatrix} R & T \\ 0 & 1 \end{bmatrix}$ as a single matrix: M

Since the transform from matrix M is isometric, $\det(M) = 1$

We can write the transformed formula for area as:

$\|(M \cdot q - M \cdot p) \times (M \cdot s - M \cdot p)\|$

$= \det(M) \cdot \|(q-p) \times (s-p)\|$

$= \|(q-p) \times (s-p)\|$

$= 1$ Q.E.D. area is preserved

Problem 1 c) Prove that under any affine transformation, the ratio of parallel line segments is invariant, but the ratio of non-parallel line segments is not invariant.

Given some vector p, an affine transformation is defined as:

$A(p) = M \cdot p + b$

For parallel line segments:

As in part a), we know $f \cdot (k_1 - k_2) = (l_1 - l_2)$, $f \in \mathbb{R}$

Then: $\frac{\|(l_1 - l_2)\|}{\|(k_1 - k_2)\|} = \frac{\| f \cdot (k_1 - k_2)\|}{\|(k_1 - k_2)\|} = f \cdot \frac{\|(k_1 - k_2)\|}{\|(k_1 - k_2)\|} = f$

Using affine transformation:

$$\frac{\|(M \cdot l_1 + b) - (M \cdot l_2 + b)\|}{\|(M \cdot k_1 + b) - (M \cdot k_2 + b)\|} = \frac{\| M \cdot (l_1 - l_2)\|}{\| M \cdot (k_1 - k_2)\|}$$

$$= \frac{\| M \cdot f \cdot (k_1 - k_2)\|}{\| M \cdot (k_1 - k_2)\|} = \frac{f \cdot \| M \cdot (k_1 - k_2)\|}{\| M \cdot (k_1 - k_2)\|} = f$$

Q.E.D. ratio of parallel lines is preserved

For non-parallel line segments:

By counterexample:

Given $k_1 = (0,0,0)$, $k_2 = (0,1,0)$, $l_1 = (0,0,0)$, $l_2 = (0,0,1)$
Forming lines k and l, each with length $= 1$

Now suppose matrix M in A applies a multiplication across axis 2, then:
$k_1 = (0,0,0)$, $k_2 = (0,2,0)$, $l_1 = (0,0,0)$, $l_2 = (0,0,1)$
Now line k has length 2, but line l still has length 1.
Thus, the ratio of non-parallel line segments has not been preserved. Q.E.D.

Problem 4 a) Show that two 3x4 camera matrices M and M' can always be reduced to the following canonical forms by an appropriate projective transformation in 3D space, which is represented by a 4x4 matrix H.

$$\hat{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \quad \hat{M}' = \begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

First, find H_0 such that $\hat{M} = M \cdot H_0$:

The most obvious value for $H_0 = \begin{bmatrix} A^{-1} & 0 \\ 0 & 0 \end{bmatrix}$, but this creates problems later with M'.

We can find a more effective answer in $H_0 = \begin{bmatrix} A^{-1} & -A^{-1}b \\ 0 & 1 \end{bmatrix}$, which allows us to cancel terms and produce same effect.

Next, we need to find H_1 such that $\hat{M} = M \cdot H_0 \cdot H_1$ and $\hat{M}' = M' \cdot H_0 \cdot H_1$

First, $M' \cdot H_0 = [A' \; b'] \begin{bmatrix} A^{-1} & -A^{-1}b \\ 0 & 1 \end{bmatrix} = [A'A^{-1} \;\; -A'A^{-1}b + b'] = \underset{3\times 4}{C}$

Let C be written in row vector form: $C = \begin{bmatrix} — & c_1 & — \\ — & c_2 & — \\ — & c_3 & — \end{bmatrix}$

Now, find H_1 such that: $C \cdot H_1 = \begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Since we need the last element of $\hat{M}' = 1$ (bottom-right), we can figure out the last column of $H_1 \cdot [0, 0, 0, \frac{1}{c_{34}}] = 1$

We can apply this method to derive the rest of H_1:

$$H_1 = \left[\begin{array}{ccc|c} & I & & 0 \\ \hline -\frac{c_{31}}{c_{34}} & -\frac{c_{32}}{c_{34}} & -\frac{c_{33}}{c_{34}} & \frac{1}{c_{34}} \end{array}\right]$$

Since H_1 must not alter $M \cdot H_0$ incorrectly, we know the top-left 3x3 is I, and the rightmost 3x1 is 0.

$M \cdot H_0 \cdot H_1 = \hat{M}$ and $M' \cdot H_0 \cdot H_1 = \hat{M}'$, $H = H_0 \cdot H_1$ (complete)

(The original transform $M \cdot H_0 = M \cdot H_0 \cdot H_1$. So we know H_1 is correct.)

Problem 4 b) Given a 4x4 matrix H representing a projective transformation in 3D space, prove that the fundamental matrices corresponding to the two pairs of camera matrices (M, M') and (MH, MH') are the same.

Given a 3D point P in the real world, we can write the value of P in each camera space as:

$p = M \cdot P, \quad p' = M' P$

When the camera space is transformed to (MH, MH') we can observe the transformation of the point P as:

$$P_H = H^{-1} P, \quad P'_H = H^{-1} P'$$

Finally,

$$\begin{aligned} p_H &= MHP_H \\ &= MHH^{-1}P \\ &= MP \\ &= p \end{aligned}$$

and

$$\begin{aligned} p'_H &= MHP'_H \\ &= MHH^{-1}P' \\ &= MP' \\ &= p' \end{aligned}$$

Therefore, the fundamental matrices corresponding to the two pairs of camera matrices are the same.

Problem 4 c) Using the conclusions from (a) and (b), derive the fundamental matrix F of the camera pair (M, M') using $a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}, b_1, b_2$. Then use the fact that F is only defined up to a scale factor to construct a seven-parameter expression for F.

From part (a), we know that $\hat{M} = MH$, $\hat{M}' = M'H$.
Combined with part (b), we know that the fundamental matrix of the camera pair (M, M') is equal to that of $(\hat{M}, \hat{M}')$.

We can the fundametal matrix F as:

$$F = \begin{bmatrix} 0 & -1 & b_2 \\ 1 & 0 & -b_1 \\ -b_2 & b_1 & 0 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} -a_{21} & -a_{22} & -a_{23}+b_2 \\ a_{11} & a_{12} & a_{13}-b_1 \\ -b_2a_{11}+b_1a_{21} & -b_2a_{12}+b_1a_{22} & -b_2a_{13}+b_1a_{23} \end{bmatrix}$$

We can factor out a_{21} to convert the matrix F into 7 parameters from 8.

$$F = \begin{bmatrix} -1 & \frac{-a_{22}}{a_{21}} & \frac{-a_{23}+b_2}{a_{21}} \\ \frac{a_{11}}{a_{21}} & \frac{a_{12}}{a_{21}} & \frac{a_{13}-b_1}{a_{21}} \\ \frac{-b_2a_{11}}{a_{21}}+b_1 & \frac{-b_2a_{12}+b_1a_{22}}{a_{21}} & \frac{-b_2a_{13}+b_1a_{23}}{a_{21}} \end{bmatrix}$$

Since F now has 7 parameters, it is properly derived.