Problem Set 1 Solution

CS231A: Computer Vision — **Chi Zhang**
Stanford University — SUID: 06116342
Spring 2017 — Date: April 22, 2017

1 Projective Geometry Problems

(a) Consider any two parallel lines k and l. Take points k_1,k_2 on k and l_1,l_2 on l. Then, $(k_1 - k_2) \times (l_1 - l_2) = 0$. If translated by vector t, then

$$(k_1 + t - k_2 - t) \times (l_1 + t - l_2 - t) = (k_1 - k_2) \times (l_1 - l_2) = 0$$

If rotated by matrix R, then we find that

$$(Rk_1 - Rk_2) \times (Rl_1 - Rl_2) = R(k_1 - k_2) \times R(l_1 - l_2) = R((k_1 - k_2) \times (l_1 - l_2)) = 0$$

Thus, parallel lines are invariant to translation and rotation.

(b) Given $pqrs$ has square unit area, then

$$\|(q - p) \times (s - p)\| = 1$$

We can multiply all points by a transformation matrix T. Since the transformation matrix is isometric, which means its determinant equals to 1, then we find that

$$\|(Tq - Tp) \times (Ts - Tp)\| = (\det T)\,\|(q - p) \times (s - p)\| = \|(q - p) \times (s - p)\| = 1$$

Thus, the same square in the camera reference system still has unit area.

(c) Consider any two parallel lines k and l. Take points k_1,k_2 on k and l_1,l_2 on l. By definition of parallel segments,

$$k_1 - k_2 = \alpha(l_1 - l_2)$$

for $\alpha \in \Re$. Thus,

$$\|k_1 - k_2\| = \alpha\,\|(l_1 - l_2)\|$$

By definition of affine transformation:

$$\|A(k_1 - k_2)\| = \|A(\alpha(l_1 - l_2))\| = \alpha\,\|A(l_1 - l_2)\|$$

We can find that the ratio of two parallel line segments is preserved under affine transformation. For the case of two non-parallel line segments, simply consider an affine transformation that only stretches by a factor of 2 in the x-direction. It is obvious that the line segment from (0, 0) to (1, 0) is stretched to a length of 2, while the line segment from (0, 0) to (0, 1) remains the same.

(d) No. Projective transformations do not necessarily preserve parallelism. Also, area is not preserved (scaling). Finally, lines can be skewed under projective transformations and thus ratios of any line segments are not preserved.

2 Affine Camera Calibration

The following code includes `compute_camera_matrix()` and `rms_error()`:

```
def compute_camera_matrix(real_XY, front_image, back_image):
    """Computes camera matrix given image and real-world coordinates.

    Args:
        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 (Z=0).
        back_image: Each row is the pixel location in the back image (Z=150).
    Returns:
        camera_matrix: The calibrated camera matrix (3x4 matrix).
    """
    img_num1 = front_image.shape[0]
    img_num2 = back_image.shape[0]

    x = np.zeros((2, img_num1+img_num2))
    for i in xrange(img_num1):
        x[:, i] = front_image[i, :].T
    for j in xrange(img_num2):
        x[:, j + img_num1] = back_image[j, :].T
    x_ones = np.ones((1, x.shape[1]))
    x = np.vstack((x, x_ones))

    X = np.zeros((2, img_num1+img_num2))
    for i in xrange(img_num1):
        X[:, i] = real_XY[i, :].T
    for j in xrange(img_num2):
        X[:, j + img_num1] = real_XY[j, :].T
    Z = np.zeros((1, X.shape[1]))
    for k in xrange(Z.shape[1]):
        if k >= img_num1:
            Z[:, k] = 150
    X = np.vstack((X, Z))
    X_ones = np.ones((1, X.shape[1]))
    X = np.vstack((X, X_ones))

    A = np.zeros((2 * (img_num1+img_num2), 8))
    for i in range(0, A.shape[0], 2):
        A[i, :] = np.hstack((X[:, i/2].T, [0, 0, 0, 0]))
        A[i + 1, :] = np.hstack(([0, 0, 0, 0], X[:, i/2].T))

    b = front_image[0].T
    for i in range(1, img_num1, 1):
        b = np.hstack((b, front_image[i].T))
    for j in range(img_num2):
        b = np.hstack((b, back_image[j].T))
    b = np.reshape(b, (2 * (img_num1 + img_num2), 1))

    p = np.linalg.inv(A.T.dot(A)).dot(A.T).dot(b)
    p = np.reshape(p, (2, -1))
    camera_matrix = np.vstack((p, [0, 0, 0, 1]))

    return camera_matrix


def rms_error(camera_matrix, real_XY, front_image, back_image):
    """Computes RMS error of points reprojected into the images.

    Args:
        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 (Z=0).
        back_image: Each row is the pixel location in the back image (Z=150).
    Returns:
        rms_error: The root mean square error of reprojecting the points back
            into the images.
    """
    img_num1 = front_image.shape[0]
    img_num2 = back_image.shape[0]

    x = np.zeros((2, img_num1+img_num2))
    for i in xrange(img_num1):
        x[:, i] = front_image[i, :].T
    for j in xrange(img_num2):
        x[:, j + img_num1] = back_image[j, :].T
    x_ones = np.ones((1, x.shape[1]))
    x = np.vstack((x, x_ones))

    X = np.zeros((2, img_num1+img_num2))
    for i in xrange(img_num1):
        X[:, i] = real_XY[i, :].T
    for j in xrange(img_num2):
        X[:, j + img_num1] = real_XY[j, :].T
    Z = np.zeros((1, X.shape[1]))
    for k in xrange(Z.shape[1]):
        if k >= img_num1:
            Z[:, k] = 150
    X = np.vstack((X, Z))
    X_ones = np.ones((1, X.shape[1]))
    X = np.vstack((X, X_ones))

    x_pred = camera_matrix.dot(X)
    diff_sqr = (x_pred - x) ** 2
    diff_sum = np.sum(np.sum(diff_sqr, axis=0))
    diff_sum /= (img_num1 + img_num2)
    rms_error = np.sqrt(diff_sum)

    return rms_error
```

(a) The camera matrix computed is

$$P = \begin{bmatrix} 0.531 & -0.018 & 0.121 & 129.7 \\ 0.048 & 0.536 & -0.103 & 44.49 \\ 0.000 & 0.000 & 0.000 & 1.000 \end{bmatrix}$$

(b) The RMS error for the camera matrix that I found in part (a) is

$$RMS_{total} = 0.99383$$

(c) No. If we are using only image, then the linear system will be rank deficient and we cannot find a unique solution. Concretely, P^TP will not have inverse matrix. Thus, we need to calibrate the camera on at least two planar surfaces.

3 Single View Geometry

The following code includes `compute_vanishing_point()`, `compute_K_from_vanishing_points()`, `compute_angle_between_planes()` and `compute_rotation_matrix_between_cameras()`:

```
def compute_vanishing_point(points):
    """Computes vanishing point given four points on parallel line.

    Args:
        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.
    """
    # construct (x1, y1) (x2, y2) (x3, y3) (x4, y4)
    x1 = points[0][0]; y1 = points[0][1];
    x2 = points[1][0]; y2 = points[1][1];
    x3 = points[2][0]; y3 = points[2][1];
    x4 = points[3][0]; y4 = points[3][1];
    # slopes
    m1 = (float)(y2 - y1) / (x2 - x1)
    m2 = (float)(y4 - y3) / (x4 - x3)
    # intercepts
    b1 = y2 - m1 * x2
    b2 = y4 - m2 * x4

    # vanishing point coordinates
    x = (b2 - b1) / (m1 - m2)
    y = m1 * ((b2 - b1)/(m1 - m2)) + b1
    vanishing_point = np.array([x, y])
    return vanishing_point


def compute_K_from_vanishing_points(vanishing_points):
    """Compute intrinsic matrix given vanishing points.

    Args:
        vanishing_points: A list of vanishing points.
    Returns:
        K: The intrinsic camera matrix (3x3 matrix).
    """
    # vanishing points used
    v1 = vanishing_points[0]
    v2 = vanishing_points[1]
    v3 = vanishing_points[2]

    # construct constraint matrix A from each pair of vanishing points
    A = np.zeros((3, 3))
    # 1 + 2
    vi = v1
    vj = v2
    A[0] = np.array([(vi[0]*vj[0]+vi[1]*vj[1]), (vi[0]+vj[0]), (vi[1]+vj[1])])

    # 1 + 3
    vi = v1
    vj = v3
    A[1] = np.array([(vi[0]*vj[0]+vi[1]*vj[1]), (vi[0]+vj[0]), (vi[1]+vj[1])])

    # 2 + 3
    vi = v2
    vj = v3
    A[2] = np.array([(vi[0]*vj[0]+vi[1]*vj[1]), (vi[0]+vj[0]), (vi[1]+vj[1])])
    # add one column of ones
    A_ones = np.ones((A.shape[0], 1))
    A = np.hstack((A, A_ones))

    # SVD
    U, s, VT = np.linalg.svd(A)
    w = VT[-1, :]
    omega = np.array([[w[0], 0, w[1]],
                      [0, w[0], w[2]],
                      [w[1], w[2], w[3]]])

    # find K matrix from omega
    KT_inv = np.linalg.cholesky(omega)
    K = np.linalg.inv(KT_inv.T)
    # normalize
    K /= K[2, 2]
    return K

def compute_angle_between_planes(vanishing_pair1, vanishing_pair2, K):
    """Compute angle between planes of the given pairs of vanishing points.

    Args:
        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.
    """
    omega_inv = K.dot(K.T)

    # a set of vanishing points on one plane
    v1 = np.hstack((vanishing_pair1[0], 1))
    v2 = np.hstack((vanishing_pair1[1], 1))

    # another set of vanishing points on the other plane
    v3 = np.hstack((vanishing_pair2[0], 1))
    v4 = np.hstack((vanishing_pair2[1], 1))

    # find two vanishing lines
    L1 = np.cross(v1.T, v2.T)
    L2 = np.cross(v3.T, v4.T)
    # find the angle between planes
    costheta = (L1.T.dot(omega_inv).dot(L2)) / (np.sqrt(L1.T.dot(omega_inv).dot(L1)) * np.sqrt
    theta = (np.arccos(costheta) / math.pi) * 180
    return theta

def compute_rotation_matrix_between_cameras(vanishing_pts1, vanishing_pts2, K):
    """Compute rotation matrix between two cameras given their vanishing points.

    Args:
        vanishing_pts1: A list of vanishing points in image 1.
        vanishing_pts2: 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.
    """
    # a set of vanishing points on one image
    v1 = np.hstack((vanishing_pts1[0], 1))
    v2 = np.hstack((vanishing_pts1[1], 1))
    v3 = np.hstack((vanishing_pts1[2], 1))

    # another set of vanishing points on the other image
    v4 = np.hstack((vanishing_pts2[0], 1))
    v5 = np.hstack((vanishing_pts2[1], 1))
    v6= np.hstack((vanishing_pts2[2], 1))

    # first image vanishing points directions
    d1 = np.linalg.inv(K).dot(v1) / np.linalg.norm(np.linalg.inv(K).dot(v1))
    d2 = np.linalg.inv(K).dot(v2) / np.linalg.norm(np.linalg.inv(K).dot(v2))
    d3 = np.linalg.inv(K).dot(v3) / np.linalg.norm(np.linalg.inv(K).dot(v3))

    # second image vanishing points directions
    dPrime1 = np.linalg.inv(K).dot(v4) / np.linalg.norm(np.linalg.inv(K).dot(v4))
    dPrime2 = np.linalg.inv(K).dot(v5) / np.linalg.norm(np.linalg.inv(K).dot(v5))
    dPrime3 = np.linalg.inv(K).dot(v6) / np.linalg.norm(np.linalg.inv(K).dot(v6))

    di = np.zeros((3, 3))
    di[:, 0] = d1.T
    di[:, 1] = d2.T
    di[:, 2] = d3.T

    diPrime = np.zeros((3, 3))
    diPrime[:, 0] = dPrime1.T
    diPrime[:, 1] = dPrime2.T
    diPrime[:, 2] = dPrime3.T
    # find rotation matrix
    R = diPrime.dot(np.linalg.inv(di))
    return R
```

(a) Refer to `compute_vanishing_point()` above.

(b) Refer to `compute_K_from_vanishing_points()` above. The computed intrinsic matrix of the camera is

$$K = \begin{bmatrix} 2594.17 & 0 & 773.290 \\ 0 & 2594.17 & 979.503 \\ 0 & 0 & 1 \end{bmatrix}$$

which is close to

$$K_{actual} = \begin{bmatrix} 2448 & 0 & 1253 \\ 0 & 2438 & 986 \\ 0 & 0 & 1 \end{bmatrix}$$

(c) No, we need orthogonal vanishing points to compute the camera intrinsic matrix. Otherwise, the set of equations will be nonlinear and difficult to solve. The minimum N is 3. Since two points yield one pair, three points yields three distinct pairs of vanishing points, which provide adequate constraints to solve this problem. If we choose only two points, then the problem will be under constrained.

(d) The simplest way is averaging. Concretely, we can use more than two parallel lines and compute the coordinates of intersections of these lines. Then we can use the average of these coordinates as a more precise estimation of the vanishing point.

(e) For detailed implementation, refer to `compute_angle_between_planes()` above. The computed angle between planes is $90.027361°$, which is approximately $90°$.

(f) For detailed implementation, refer to `compute_rotation_matrix_between_cameras()` above. The computed rotation matrix between cameras is

$$R = \begin{bmatrix} 0.962 & 0.049 & -0.158 \\ -0.010 & 1.007 & 0.046 \\ 0.189 & -0.069 & 1.005 \end{bmatrix}$$

the rotation angles are $\theta_x = -2.605°$, $\theta_y = -8.919°$ and $\theta_z = -2.932°$, respectively.

4 Fundamental Matrix

(a) It is not easy to find H_0, since it transforms M to $\hat{M}$:

$$\begin{bmatrix} A & b \end{bmatrix} \xrightarrow{H_0} \begin{bmatrix} I & 0 \end{bmatrix}$$

then

$$H_0 = \begin{bmatrix} A^{-1} & -A^{-1}b \\ 0 & 1 \end{bmatrix}$$

For M' and $\hat{M}'$:

$$M'H_0 = \begin{bmatrix} A' & b' \end{bmatrix} \begin{bmatrix} A^{-1} & -A^{-1}b \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} A'A^{-1} & -A'A^{-1}b + b' \end{bmatrix}$$

and dimension of $M'H_0$ is 3×4, then we suppose that

$$M'H_0 = \begin{bmatrix} m'_{11} & m'_{12} & m'_{13} & m'_{14} \\ m'_{21} & m'_{22} & m'_{23} & m'_{24} \\ m'_{31} & m'_{32} & m'_{33} & m'_{34} \\ m'_{41} & m'_{42} & m'_{43} & m'_{44} \end{bmatrix}$$

So now, the task is to find H_1 that satisfies

$$\begin{bmatrix} m'_{11} & m'_{12} & m'_{13} & m'_{14} \\ m'_{21} & m'_{22} & m'_{23} & m'_{24} \\ m'_{31} & m'_{32} & m'_{33} & m'_{34} \\ m'_{41} & m'_{42} & m'_{43} & m'_{44} \end{bmatrix} \xrightarrow{H_1} \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The last column of H_1, say [a, b, c, d]^T is subject to

$$am'_{31} + bm'_{32} + cm'_{33} + dm'_{34} = 1$$

one simple solution is [0, 0, 0, $\frac{1}{m'_{34}}$]^T. Similarly, it is easy to find

$$H_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\frac{m'_{31}}{m'_{34}} & -\frac{m'_{32}}{m'_{34}} & -\frac{m'_{33}}{m'_{34}} & \frac{1}{m'_{34}} \end{bmatrix}$$

To justify the answer,

$$MH_0H_1 = MI = \hat{M}$$

which means H_1 does not effect the reduction of M to $\hat{M}$.

(b) Say that the image point correspondence $x \leftrightarrow x'$ derives from a 3D point X under camera space (M, M') as

$$x = MX$$

$$x' = M'X$$

If the camera space is transformed to $(MH, M'H)$, the 3D point then transforms as

$$X_0 = H^{-1}X$$

$$X'_0 = H^{-1}X'$$

$$x_0 = MHX_0 = MHH^{-1}X = MX = x$$

and likewise with X'_0 still get us the same image points. Thus, the fundamental matrices corresponding to the two pairs of camera matrices (M, M') and $(MH, M'H)$ are the same.

(c) Based on conclusion from (b), the fundamental matrix F of the camera pair (M, M') is equivalent to F' of the camera pair $(MH_0H_1, M'H_0H_1)$ (based on conclusion from (a), which is $(\hat{M}, \hat{M}'))$.

$$F = \begin{bmatrix} 0 & -1 & b_2 \\ 1 & 0 & -b_1 \\ -b_2 & b_1 & 0 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -a_{21} & -a_{22} & -a_{23} \\ a_{11} & a_{12} & a_{13} \\ -b_2a_{11} + b_1a_{21} & -b_2a_{12} + b_1a_{22} & -b_2a_{13} + b_1a_{23} \end{bmatrix}$$

There are eight parameters in the above matrix, let all elements divide by a_{21} to factor out it.

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

Now, F is expressed in a seven-parameter matrix.