Cork Institute of Technology

COMP9074 MACHINE VISION: ASSIGNMENT 2

PREPARED BY: MARCUS WONG YEW HON

STUDENT ID: R00183595

Table of Contents

Task 1: Pre-processing

Subtask A

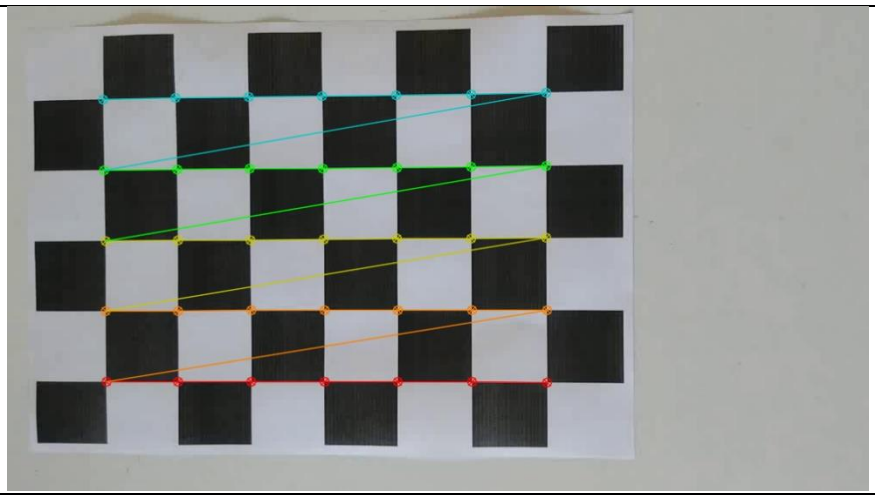
Load calibration images and video file

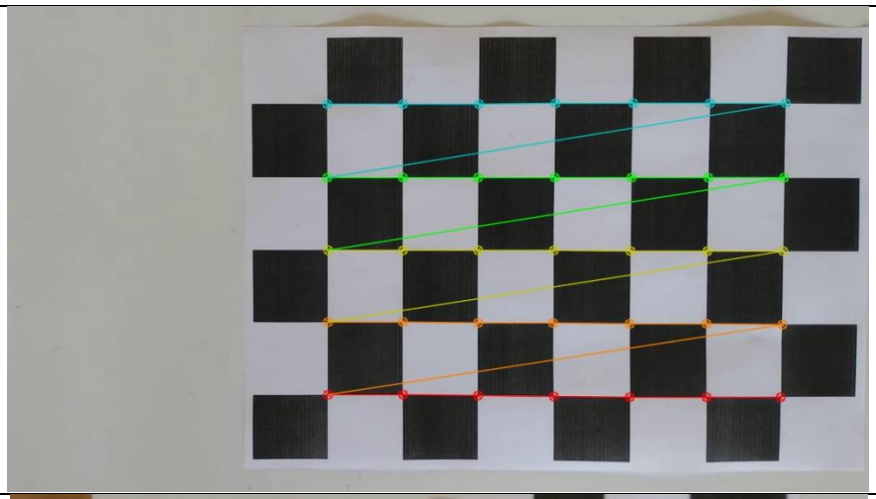
```
def getInputData():
    path = os.path.dirname(os.path.realpath(__file__))
    file_names = [filename for filename in os.listdir(path+'\\Assignment_MV_02_ca
libration\\') if filename.endswith("png")]
    chessImages = [os.path.join(path+'\\Assignment_MV_02_calibration\\', fn) for
fn in file_names]
    vid_name = "Assignment_MV_02_video.mp4"
    video = os.path.join(path, vid_name)
    return chessImages, video
```

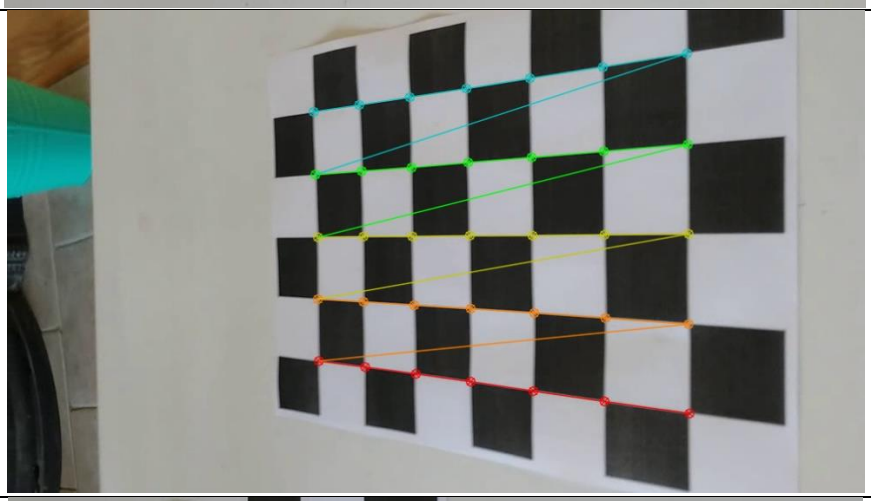
Extract checkerboard corners to subpixel accuracy

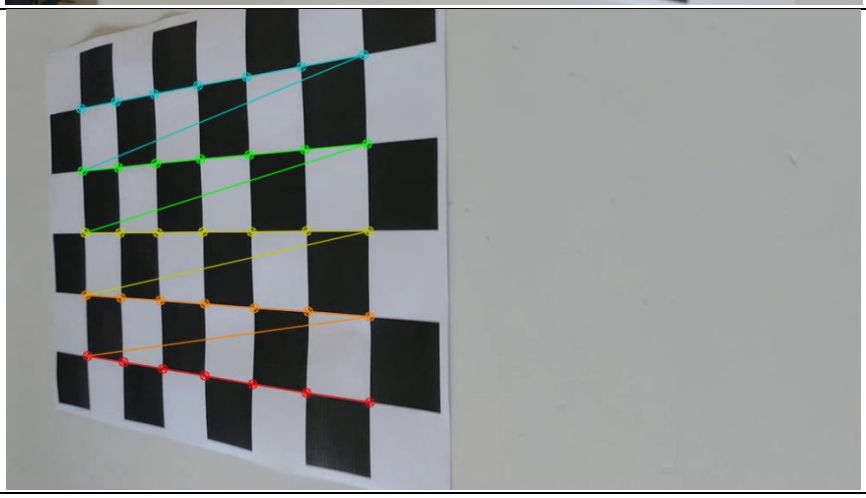
```
ret, corners = cv2.findChessboardCorners(gray, (7,5), None)

        if ret == True:
            objpoints.append(objp)
            # get subpixel accurate corner points
            corners2 = cv2.cornerSubPix(gray,corners, (11,11), (-1,-1), criteria)
            imgpoints.append(corners)
            # Draw and display the corners
            cv2.drawChessboardCorners(img, (7,5), corners2, ret)
```

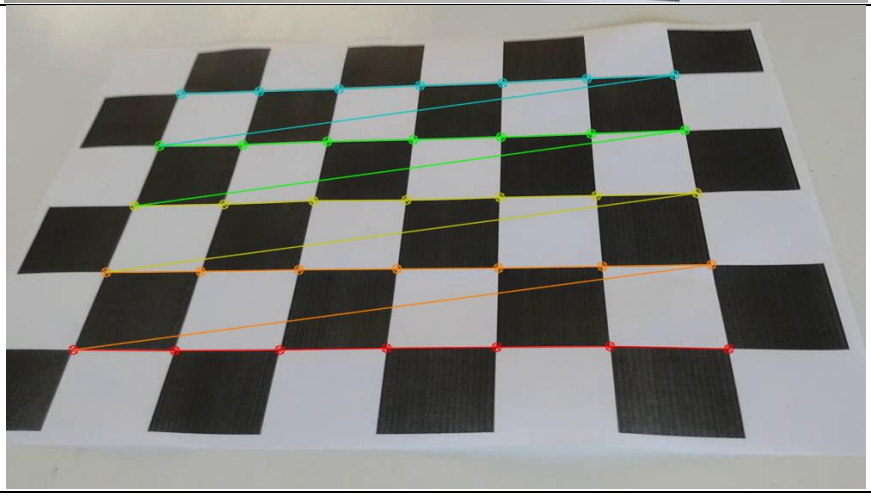
Results:









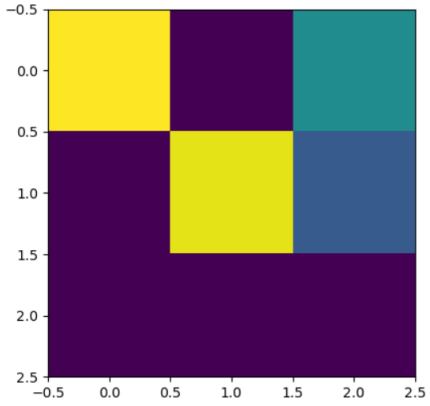


Subtask B

Obtain camera calibration matrix K

```
ret, mtx_K, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, gray.shape[::-1], None, None)
return mtx_K
```

Output of matrix K:




```
[[994.93741694   0.         485.13219579]
 [  0.         953.82995345 286.16775074]
 [  0.           0.           1.        ]]
```

Subtask C

Identify good features to track for first frame of input video. Refine to subpixel accuracy.

```
def firstFrameFeatures(video):
    # extract 1st frame
    images = extract_frames(video, [1])
    img = images[1].copy()
    new_img = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    # obtain features from extracted frame
    corners = cv2.goodFeaturesToTrack(new_img, 200, 0.3, 7)
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 100, 0.001)
    # get features with subpixel accuracy
    p0 = cv2.cornerSubPix(new_img,np.float32(corners),(5,5),(-1,-1),criteria)
    # visualize feature points
    for i,points in enumerate(p0):
        x,y = points[0]
        cv2.circle(img,(x,y), 5, (0,255,0), -1)
    cv2.imshow("", img)
    cv2.waitKey()
    cv2.destroyAllWindows()
    saveImg("firstFrameFeatures.png", img)
    return img
```

First frame features visualized:

Subtask D

Track features across whole image sequence and refine coordinates to subpixel accuracy

```
def featureTracking(video):
    stream = cv2.VideoCapture(video)
    while stream.isOpened():
        ret,img= stream.read()
        if ret:
            new_img = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
            p0 = cv2.goodFeaturesToTrack(new_img, 200, 0.3, 7)
            # refine features to subpixel accuracy
            criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 100,
0.001)
            p0 = cv2.cornerSubPix(new_img,np.float32(p0),(5,5),(-1,-
1),criteria)
            break

    # initialise tracks
    index = np.arange(len(p0))
    tracks = {}
    for i in range(len(p0)):
        tracks[index[i]] = {0:p0[i]}

    frame = 0
    frame_img=[]
    while stream.isOpened():
        ret,img= stream.read()
        if not ret:
            break

        frame += 1
        old_img = new_img
        new_img = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

        # calculate optical flow
        if len(p0)>0:
            p1, st, err  = cv2.calcOpticalFlowPyrLK(old_img, new_img, p0, None)
            # visualise points
            for i in range(len(st)):
                if st[i]:
                    cv2.circle(img, (p1[i,0,0],p1[i,0,1]), 2, (0,0,255), 2)
                    cv2.line(img, (p0[i,0,0],p0[i,0,1]), (int(p0[i,0,0]+(p1[i][0,
0]-p0[i,0,0])*5),int(p0[i,0,1]+(p1[i][0,1]-
p0[i,0,1])*5)), (0,0,255), 2)

            p0 = p1[st==1].reshape(-1,1,2)
            index = index[st.flatten()==1]
```

Task 2: Fundamental Matrix

Subtask A

Extract and visualize the feature tracks obtained in task 1.

Establish coordinate correspondence between first frame and last frame.

```
def getCorrespondences(tracks, frame1, frame2):
    correspondences = []
    # loop through tracks to get frame correspondences
    for track in tracks:
        if (frame1 in tracks[track]) and (frame2 in tracks[track]):
            x1 = [tracks[track][frame1][0,0],tracks[track][frame1][0,1],1]
            x2 = [tracks[track][frame2][0,0],tracks[track][frame2][0,1],1]
            correspondences.append((np.array(x1), np.array(x2)))
    return correspondences
```

Subtask B

Calculate mean and standard deviations for correspondences coordinates.

```
# calculate mean and standard deviation for each correspondence
    mean = np.squeeze(np.mean(np.array([correspondences]), axis=1))
    std = np.squeeze(np.std(np.array([correspondences]), axis=1))
```

Normalization homographies:

```
# get T and T_
    T = np.array([[1/std[0][0], 0, -mean[0][0]/std[0][0]],
                  [0,1/std[0][1], -mean[0][1]/std[0][1]],
                  [0,0,1]])

    T_ = np.array([[1/std[1][0], 0, -mean[1][0]/std[1][0]],
                  [0,1/std[1][1], -mean[1][1]/std[1][1]],
                  [0,0,1]])
```

Subtask C

Select eight feature correspondence at random and build a Kron matrix to calculate the fundamental matrix.

```
# obtain 8 random correspondences
        random_idx = random.sample(range(len(correspondences)), 8)
        random_correspondences = [correspondences[idx] for idx in random_idx]

        # normalize the 8 random correspondences
        # build matrix using said correspondences
        A = np.zeros((0,9))
        for x1,x2 in random_correspondences:
            x1n = np.matmul(T,x1)
            x2n = np.matmul(T_,x2)
            ai = np.kron(x1n.T,x2n)
            A = np.append(A,[ai],axis=0)
```

Subtask D

Use the 8-point DLT algorithm to calculate the estimated fundamental matrix F-hat. Ensure F-hat is singular and calculate the actual fundamental matrix using the normalization homographies from subtask B with F-hat.

```
      U,S,V = np.linalg.svd(A)
      F = V[8,:].reshape(3,3).T

      U,S,V = np.linalg.svd(F)
      # calculate fundamental matrix
      F = np.matmul(U,np.matmul(np.diag([S[0],S[1],0]),V))
      F = np.matmul(T_.T, np.matmul(F, T))
```

Subtask E

Get remaining unselected feature correspondences. Calculate the value of model equation and variance using the point observation covariance matrix *Cxx*.

```
# get unselected correspondences
new_correspondences = [element for i, element in enumerate(correspondences) if i
not in random_idx]
x = np.squeeze(np.array([new_correspondences]))[:,0]
x_ = np.squeeze(np.array([new_correspondences]))[:,1]

# point observation covariance matrix
Cxx = np.array([[1,0,0],
                [0,1,0],
                [0,0,0]])

count_outliers = 0
accumulate_error = 0
inliers = []
# loop through unselected correspondences
for x1,x2 in new_correspondences:
    # calculate gi and variance
    value = np.matmul(np.matmul(x2.T, F), x1)

    variance = np.matmul(np.matmul(np.matmul(np.matmul(x2.T, F), Cxx), F.T), x2)
    variance += np.matmul(np.matmul(np.matmul(np.matmul(x1.T, F.T), Cxx), F), x1)
```

Subtask F

Determine for each correspondence if they are outliers by calculating the test statistic with an outlier threshold of 6.635. Sum the test statistics over all inliers.

```
# threshold the test statistics
            if test_stat > 6.635:
                # count outliers
                count_outliers += 1
                # record outliers
                outliers.append((x1,x2))
            else:
                # sum the test statistics for inliers
                accumulate_error += test_stat
                # record inliers
                inliers.append((x1,x2))
```

Subtask G

Repeat the above procedure 10000 times. Select the fundamental matrix and remove all outliers for the selection of eight points which yielded the least number of outliers. Break ties by looking at the sum of the test statistic over the inliers.

```
# to record lowest amount of outliers
        if count_outliers < best_outliers:
            best_error = accumulate_error
            best_outliers = count_outliers
            # get fundamental matrix and inliers for lowest amount of outliers
            final_Fmat = F
            final_inliers = inliers
            final_outliers = outliers
        # break ties for number of outliers
        elif count_outliers ==  best_outliers:
            # compare sum of test statistics when tied for number of outliers
            if accumulate_error < best_error:
                best_error = accumulate_error
                best_outliers = count_outliers
                # get fundamental matrix and inliers
                final_Fmat = F
                final_inliers = inliers
                final_outliers = outliers
```

Subtask H

Indicate outliers (red) and inliers (blue) for first and last frame:

First Frame

Last Frame

Calculate epipoles:

```
def calculate_epipoles(F):
    U,S,V = np.linalg.svd(F)
    e1 = V[2,:]

    U,S,V = np.linalg.svd(F.T)
    e2 = V[2,:]

    return e1,e2
```

Output coordinates of epipoles:

[0.80958472 0.58700027 0.00180572]

[-0.80917992 -0.58755817 -0.00180669]

Task 3: Essential Matrix

Subtask A

```python
def calculateEssentialMatrix(matrix_K, F):
    # calculate essential mat
    E = np.matmul(np.matmul(matrix_K.T, F), matrix_K)

    # decompose E
    U,S,V = np.linalg.svd(E)

    # ensure positive determinants
    if np.linalg.det(U)<0:
        U[:,2] *= -1
    if np.linalg.det(V)<0:
        V[:,2] *= -1
    return U,V
```

Subtask B

Determine the four potential combinations of rotation matrices $\boldsymbol{R}$ and translation vector $\boldsymbol{t}$ between the first and the last frame. Assume the camera was moving at 50km/h and that the video was taken at 30fps to determine the scale of the baseline $\boldsymbol{t}$ in meters.

```python
def getRotate_Translate(U,V):
    W = np.array([[0,-1,0],
                  [1,0,0],
                  [0,0,1]])

    Z = np.array([[0,1,0],
                  [-1,0,0],
                  [0,0,0]])

    # assume 50km/h and 30fps
    B = (50000/3600) * (30/30)

    # skew of R.T and t
    skew_1 = B * np.matmul(U,np.matmul(Z,U.T))
    skew_2 = -(skew_1)

    # get R.T * t
    X_POS = [skew_1[2,1], skew_1[0,2], skew_1[1,0]]
    X_NEG = [skew_2[2,1], skew_2[0,2], skew_2[1,0]]

    # R1,R2
    R1 = np.matmul(U, np.matmul(W, V.T))
    R2 = np.matmul(U, np.matmul(W.T, V.T))

    # 4 translations matrices
    t1= np.matmul(np.linalg.inv(R1), X_POS)
    t2= np.matmul(np.linalg.inv(R1), X_NEG)
    t3= np.matmul(np.linalg.inv(R2), X_POS)
    t4= np.matmul(np.linalg.inv(R2), X_NEG)

    # 4 possible combinations
    combo = [(R1,t1), (R1,t2), (R2,t2), (R2,t2)]

    return combo
```

Subtask C

```
def get3dpoints(combo, inliers, matrix_K):
    print ("length of inliers:",len(inliers))
    # 3d point coords
    x,y,z = [],[],[]
    best_solution = len(inliers)+1
    # for each translation and rotation combination
    for c in combo:
        r,t = c[0], c[1]
        count = 0
        # loop through inliers
        for x1,x2 in inliers:
            m = np.matmul(np.linalg.inv(matrix_K),x1)
            m_ = np.matmul(np.linalg.inv(matrix_K),x2)
            # linear equation set up
            LHS = np.array([[np.matmul(m.T, m), np.matmul(-m.T, np.mat-
mul(r.T, m_))],
                            [np.matmul(m.T, np.matmul(r.T, m_)), np.matmul(-
m.T, m_)]])
            RHS = np.array([np.matmul(t.T, m),
                            np.matmul(t.T, np.matmul(r.T, m_))])
            # solve linear equation
            # returns 2 element array containing results from linear equation
            res = np.matmul(np.linalg.inv(LHS), RHS)
            lambd, mue = res[0], res[1]
            # if points are in front of both cameras
            if lambd > 0 and mue > 0:
                count+=1
                # calculate the 3d points for both frame correspondence
                x_lambd = lambd * m
                x_mue = t + (mue*np.matmul(r.T, m_))
                # obtain final 3d point by averaging the distance be-
tween xlambda and xmue
                x_final = (x_lambd+x_mue) / 2
                x.append(x_final[0])
                y.append(x_final[1])
                z.append(x_final[2])
        # get 3d coordinates and translation vector that yielded the best solu-
tion
        if count < best_solution:
            best_solution = count
            best_x = x
            best_y = y
            best_z = z
            best_t = t
```

Subtask D

Create a 3d plot to show the two camera centers (blue) and all 3d points (red).

```
def plot3d(x,y,z, t):
    import matplotlib.pyplot as plt

    fig = plt.figure()
    ax = plt.axes(projection='3d')

    # plot 3d points
    ax.scatter3D(x, y, z, c='r')

    # plot 1st and 2nd camera center
    ax.scatter3D(0,0,0, c='b')
    ax.scatter3D(t[0],t[1],t[2], c='b')

    plt.show()
```

Output: