Homework 3: AR Tracking and 3D Reconstruction with OpenCV

Overview:

For this homework you will be implementing pose estimation of a known image using OpenCV. You will then adapt this pose estimation algorithm to compute a sparse point cloud representation of a scene. This homework is intended to give you a basic introduction to computer vision as it relates to augmented reality. This includes concepts such as camera calibration, feature matching, perspective projection, and 3d point cloud estimation.

In this homework you will be guided through most of the steps to implement AR pose estimation of a known image, but you will be asked to implement one basic function based on the concepts we introduced in class. You will then use what you have learned to develop a system for sparse point cloud reconstruction of a scene using your mobile device. These steps represent a simplified set of functions that form the basis of modern AR tracking.

Note: This document was written using OpenCV 3.4.7. Some changes may be required for alternate versions of OpenCV.

Deliverables:

1. Video:

You will make a 2 minute video showing off your implementation. You should verbally describe, at a very high level, the concepts used to implement the image pose tracking and 3d reconstruction. You must also include captions corresponding to the audio. This will be an important component of all your homework assignments and your final project so it is best you get this set up early.

2. Code:

You will also need to upload your project folder to the Github Classroom assignment:

Before You Start:

For this homework you will need Python and OpenCV. After installing Python, to install OpenCV:
In command prompt/ terminal run:
`pip install opencv-contrib-python`
(you may need to add sudo for unix systems).

Part 1 - Camera Calibration:

Camera calibration is a necessary step for many if not most computer vision algorithms. In a basic sense, it allows us to account for the fact that every camera is different due to different resolutions and different lens characteristics.

For this section we will be performing camera calibration using a standard checkerboard. This is the most common method of performing camera calibration. Download and print out the checkerboard from here:
https://raw.githubusercontent.com/opencv/opencv/master/doc/pattern.png

You will want to tape this checkerboard to a rigid flat movable surface (a clipboard is a common solution). Any distortion in the surface or the paper will result in an inaccurate camera calibration.

We will compute both our camera matrix (also called the camera intrinsic matrix), and the distortion coefficients. The camera matrix represents the translation and scaling introduced by the camera. This accounts both for the focal length/ field of view of the lens as well as the representation of image points as pixels. As a reminder this is defined as:

$$camera\ matrix = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Where (fx, fy) are the focal length parameters, and (cx, cy) is the center point.

The distortion coefficients represent a nonlinear distortion introduced by the lens. This distortion can be modeled as having a radial and tangential component. The radial

component causes straight lines to appear curved, with the effect becoming more pronounced the further points are from the image center. Radial distortion can be modeled as:

$$x_{distorted} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$
$$y_{distorted} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

Similarly tangential distortion is represented as:

$$x_{distorted} = x + [2p_1 xy + p_2(r^2 + 2x^2)]$$
$$y_{distorted} = y + [p_1(r^2 + 2y^2) + 2p_2 xy]$$

These equations are only provided for your reference, you do not need to memorize or understand them.

The first step towards camera calibration is to collect images. Create a python file named CalibrationHelpers.py and add the following code (as always be sure to read the code and comments to understand what is going on):

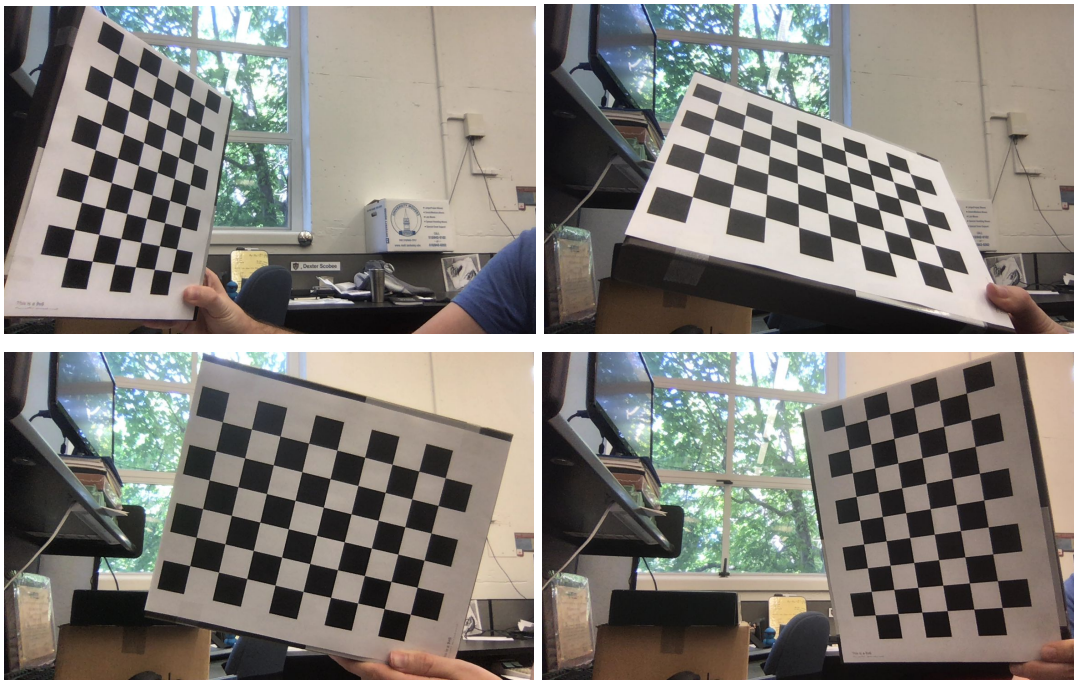
```
import cv2
import numpy as np
import glob

# This function records images from the connected camera to specified directory
# when the "Space" key is pressed.
# directory: should be a string corresponding to the name of an existing
# directory
def CaptureImages(directory):
    # Open the camera for capture
    # the 0 value should default to the webcam, but you may need to change this
    # for your camera, especially if you are using a camera besides the default
    cam = cv2.VideoCapture(0)
    img_counter = 0
    # Read until user quits
    while True:
        ret, frame = cam.read()
        if not ret:
            break
        # display the current image
        cv2.imshow("Display", frame)
        # wait for 1ms or key press
```

```
        k = cv2.waitKey(1) #k is the key pressed
        if k == 27 or k==113:  #27, 113 are ascii for escape and q respectively
            #exit
            break
        elif k == 32: #32 is ascii for space
            #record image
            img_name = "calib_image_{}.png".format(img_counter)
            cv2.imwrite(directory+'/'+img_name, frame)
            print("Writing: {}".format(directory+'/'+img_name))
            img_counter += 1
    cam.release()
```

This code records an image whenever the space key is pressed. Open python and run:

```
import CalibrationHelpers as calib
mkdir calibration_data
calib.CaptureImages('calibration_data')
```

Reposition your checkerboard at different positions and angles in the image. Press space at each location to capture an image. Ensure the full checkerboard is always visible. Collect around 20 images (only 10 are required, but 20 is recommended as some may not detect the full checkerboard properly). Examples:



Once you have collected a set of images it is time to perform camera calibration. Add the following code to CalibrationHelpers.py:

```
# This function calls OpenCV's camera calibration on the directory of images
# created above.
# Returns the following values
# intrinsics: the current camera intrinsic calibration matrix
# distortion: the current distortion coefficients
# roi: the region of the image with full data
# new_intrinsics: the intrinsic calibration matrix of an image after
# undistortion and roi cropping
def CalibrateCamera(directory,visualize=False):
    # termination criteria
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)
    # here we only care about computing the intrinsic parameters of the camera
    # and not the true positions of the checkerboard, so we can do everything
    # up to a scale factor, this means we can prepare our object points as
    # (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0) representing the coordinates
    # of the corners in the checkerboard's local coordinate frame
    # if we cared about exact position we would need to scale these according
    # to the true size of the checkerboard
    objp = np.zeros((9*6,3), np.float32)
    objp[:,:2] = np.mgrid[0:9,0:6].T.reshape(-1,2)
    # Set up arrays to store object points and image points from all the images
    # Here the image points are the 2d positions of the corners in each image
    # the object points are the true 3d positions of the corners in the
    # checkerboards coordinate frame
    objpoints = [] # 3d point in the checkerboard's coordinate frame
    imgpoints = [] # 2d points in image plane.
    # Grab all images in the directory
    images = glob.glob(directory+'/*.png')
    for fname in images:
        # read the image
        img = cv2.imread(fname)
        # convert to grayscale (this simplifies corner detection)
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        # Find the chess board corners
        ret, corners = cv2.findChessboardCorners(gray, (9,6), None)
        # If found, add object points, image points (after refining them)
        # This ensures that only images where all corners could be detected are
        # used for calibration
        if ret == True:
            # the object points in the checkerboard frame are always the same
            # so we just duplicate them for each image
            objpoints.append(objp)
            # refine corner locations, initial corner detection is performed by
```

```
            # sliding a convultional filter across the image, so accuracy is
            # at best 1 pixel, but from the image gradient we can compute the
            # location of the corner at sub-pixel accuracy
            corners2 = \
                cv2.cornerSubPix(gray,corners, (11,11), (-1,-1), criteria)
            # append the refined pixel location to our image points array
            imgpoints.append(corners2)
            # if visualization is enabled, draw and display the corners
            if visualize==True:
                cv2.drawChessboardCorners(img, (9,6), corners2, ret)
                cv2.imshow('Display', img)
                cv2.waitKey(500)
    # Perform camera calibration
    # Here I have fixed K3, the highest order distortion coefficient
    # This simplifies camera calibration and makes it easier to get a good
    # result, however this is only works under the assumption that your camera
    # does not have too much distortion, if your camera is very wide field of
    # view, you should remove this flag
    ret, intrinsics, distortion, rvecs, tvecs = \
        cv2.calibrateCamera(objpoints, imgpoints, gray.shape[::-1], None, \
                            None,flags=cv2.CALIB_FIX_K3,criteria=criteria)
    # print error if calibration unsuccessful
    if not ret:
        print("Calibration failed, recollect images and try again")
    # if successful, compute an print reprojection error, this is a good metric
    # for how good the calibration is. If your result is greater than 1px you
    # should probably recalibrate
    total_error = 0
    for i in range(len(objpoints)):
        # project the object points onto each camera image and compare
        # against the detected image positions
        imgpoints2, _ = cv2.projectPoints(objpoints[i], rvecs[i], tvecs[i], \
                                          intrinsics, distortion)
        error = cv2.norm(imgpoints[i], imgpoints2, cv2.NORM_L2)/len(imgpoints2)
        total_error += error
    print( "mean error: {}".format(total_error/len(objpoints)) )
    # compute the region for where we have full information and the resulting
    # intrinsic calibration matrix
    h,  w = img.shape[:2]
    new_intrinsics, roi = cv2.getOptimalNewCameraMatrix(intrinsics, \
                                                        distortion, (w,h), 1,\
                                                        (w,h))
    # return only the information we will need going forward
    return intrinsics, distortion, roi, new_intrinsics

# This function will save the calibration data to a file in the specified
# directory
```

```
def SaveCalibrationData(directory, intrinsics, distortion, new_intrinsics, \
                        roi):
    np.savez(directory+'/calib', intrinsics=intrinsics, distortion=distortion,\
             new_intrinsics = new_intrinsics, roi=roi)

# This function will load the calibration data from a file in the specified
# directory
def LoadCalibrationData(directory):
    npzfile = np.load(directory+'/calib.npz')
    return npzfile['intrinsics'], npzfile['distortion'], \
            npzfile['new_intrinsics'], npzfile['roi']
```

To run the calibration, open python and run:

```
import CalibrationHelpers as calib
intrinsics, distortion, roi, new_intrinsics =\
calib.CalibrateCamera('calibration_data',true)
```

This will go through all the images and display the detected corners (if corner detection was successful). Visually inspect the images as they appear to ensure that the corners appear to be detected in the correct location.

This function will also print out the reprojection error. The reprojection error is a great metric for determining the quality of the resulting calibration. If your reprojection error is greater than 1 pixel you should definitely recollect your images and recalibrate.

If your calibration looks good, go ahead and save your calibration:

```
calib.SaveCalibrationData('calibration_data', intrinsics, distortion, new_intrinsics,
                          roi)
```

Part 2 - AR Image Tracking:

For this we will be tracking a known image and computing the camera pose relative to this image in the scene. While in theory any 2d image with features can be used, we have prepared an image that we know will work well. Download this image and place it in your code directory: image

You will also need to print out this image, when printing this image be sure to set the scale to 100%. This ensures that the image will be printed at the proper size (10cm x 10cm). As with the checkerboard above, you should tape this to a rigid flat movable surface.

The first thing we are going to look at is feature detection. Features are simply points in the scene that we expect to be able to repeatedly find in multiple images. The corners we used during camera calibration are one example of such features. There are quite a few different types of features, even within OpenCV, each with their own benefits and drawbacks. For this we will use BRISK features.

Create a new file named ARImagePoseTracker.py and add the following code to it:

```
import cv2
import numpy as np
import CalibrationHelpers as calib

# Load the reference image that we will try to detect in the webcam
reference = cv2.imread('ARTrackerImage.jpg',0)
RES = 480
reference = cv2.resize(reference,(RES,RES))

# create the feature detector. This will be used to find and describe locations
# in the image that we can reliably detect in multiple images
feature_detector = cv2.BRISK_create(octaves=5)
# compute the features in the reference image
reference_keypoints, reference_descriptors = \
        feature_detector.detectAndCompute(reference, None)
# make image to visualize keypoints
keypoint_visualization = cv2.drawKeypoints(
        reference,reference_keypoints,outImage=np.array([]))

# display the image
cv2.imshow("Keypoints",keypoint_visualization)
# wait for user to press a key before proceeding
cv2.waitKey(0)
```

Run this file. What you should see is the reference image with hundreds of little circles overlaid on it. Each of these circles represents a feature that we are hoping to find when observing the image in the physical world.

Change the drawKeypoints command to:

```
keypoint_visualization = cv2.drawKeypoints(
        reference,reference_keypoints,outImage=np.array([]),
        flags = cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
```

Rerun the file. Now on the reference image, each little circle is replaced with a larger circle. This larger circle is representative of the size of the feature in the image. This size corresponds to the image patch that was used to compute the feature descriptor. The feature descriptor is a compact description of the area around the detected feature that can be used to identify when the same feature appears in multiple images.

In reality the feature descriptor will change slightly between images as the camera perspective or lighting conditions change. This means that the best we can do to identify repeated features is to find the most similar features between images. This is what we will do in the next section of code. In ARImagePoseTracker.py, remove the cv2.waitKey (so we don't have to wait every time we run this code), and add the following:

```
# create the matcher that is used to compare feature similarity
# Brisk descriptors are binary descriptors (a vector of zeros and 1s)
# Thus hamming distance is a good measure of similarity
matcher = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

# Load the camera calibration matrix
intrinsics, distortion, new_intrinsics, roi = \
        calib.LoadCalibrationData('calibration_data')

# initialize video capture
# the 0 value should default to the webcam, but you may need to change this
# for your camera, especially if you are using a camera besides the default
cap = cv2.VideoCapture(0)

while True:
    # read the current frame from the webcam
    ret, current_frame = cap.read()

    # ensure the image is valid
    if not ret:
        print("Unable to capture video")
        break

    # undistort the current frame using the loaded calibration
    current_frame = cv2.undistort(current_frame, intrinsics, distortion, None,\
                                  new_intrinsics)
    # apply region of interest cropping
    x, y, w, h = roi
    current_frame = current_frame[y:y+h, x:x+w]

    # detect features in the current image
    current_keypoints, current_descriptors = \
```

```
        feature_detector.detectAndCompute(current_frame, None)

    # match the features from the reference image to the current image
    matches = matcher.match(reference_descriptors, current_descriptors)
    # matches returns a vector where for each element there is a
    # query index matched with a train index. I know these terms don't really
    # make sense in this context, all you need to know is that for us the
    # query will refer to a feature in the reference image and train will
    # refer to a feature in the current image

    # create a visualization of the matches between the reference and the
    # current image
    match_visualization = cv2.drawMatches(reference, reference_keypoints,
current_frame,
                          current_keypoints, matches, 0,
                          flags=
                          cv2.DRAW_MATCHES_FLAGS_NOT_DRAW_SINGLE_POINTS)
    cv2.imshow('matches',match_visualization)
    k = cv2.waitKey(1)
    if k == 27 or k==113:  #27, 113 are ascii for escape and q respectively
        #exit
        break
```

When you run this code you should see the reference image alongside your current image in the webcam, along with a couple hundred lines drawn between the two images. Each line represents a correspondence between a feature in the reference image to its most similar feature in the current image.

If you hold up your printed out reference image, you should see that many (but not all) of the lines correspond to this image. Due to noise in the camera, as well as different lighting and perspective, not all features will be correctly matched to their appropriate counterpart. Later you will implement a method to determine which features are correctly matched between the images. For now, try moving and rotating your printed reference image and observe how this affects the feature matches.

The next step is to compute pose from these matches. First let’s comment out our match visualization code (from “match_visualization=” onwards), as drawing matching takes quite a bit of computation.

OpenCV has several methods for computing camera pose from a set points and matches such as PnP and FindFundamentalMat. We will be using a special version that uses the prior knowledge that all our features lie on the same plane. This type of pose

estimation is called homography. Knowing how to compute the homography is beyond the scope of this class, but there are many online resources if you are interested.

To compute pose from homography add the following to your file:

```
    # set up reference points and image points
    # here we get the 2d position of all features in the reference image
    referencePoints = np.float32([reference_keypoints[m.queryIdx].pt \
                                  for m in matches])
    # convert positions from pixels to meters
    SCALE = 0.1 # this is the scale of our reference image: 0.1m x 0.1m
    referencePoints = SCALE*referencePoints/RES

    imagePoints = np.float32([current_keypoints[m.trainIdx].pt \
                                  for m in matches])
    # compute homography
    ret, R, T = ComputePoseFromHomography(new_intrinsics,referencePoints,
                                          imagePoints)
    if(ret):
        print(T)
```

And to the top of your file (just below the import commands) add:

```
# This function takes in an intrinsics matrix, and two sets of 2d points
# if a pose can be computed it returns true along with a rotation and
# translation between the sets of points.
# returns false if a good pose estimate cannot be found
def ComputePoseFromHomography(new_intrinsics, referencePoints, imagePoints):
    # compute homography using RANSAC, this allows us to compute
    # the homography even when some matches are incorrect
    homography, mask = cv2.findHomography(referencePoints, imagePoints,
                                          cv2.RANSAC, 5.0)
    # check that enough matches are correct for a reasonable estimate
    # correct matches are typically called inliers
    MIN_INLIERS = 30
    if(sum(mask)>MIN_INLIERS):
        # given that we have a good estimate
        # decompose the homography into Rotation and translation
        # you are not required to know how to do this for this class
        # but if you are interested please refer to:
        # https://docs.opencv.org/master/d9/dab/tutorial_homography.html
        RT = np.matmul(np.linalg.inv(new_intrinsics), homography)
        norm = np.sqrt(np.linalg.norm(RT[:,0])*np.linalg.norm(RT[:,1]))
        RT = -1*RT/norm
        c1 = RT[:,0]
        c2 = RT[:,1]
        c3 = np.cross(c1,c2)
        T = RT[:,2]
```

```
        R = np.vstack((c1,c2,c3)).T
        W,U,Vt = cv2.SVDecomp(R)
        R = np.matmul(U,Vt)
        return True, R, T
    # return false if we could not compute a good estimate
    return False, None, None
```

Now when you run this code you should see a translation vector printed in the console whenever you hold up your image. This indicates the 3D position of the camera with respect to the top left corner of your image.

Now we would like to actually render an AR object using this pose. To do this, replace:

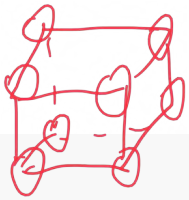
```
    if(ret):
        print(T)
```

At the bottom of your code with:

```
    render_frame = current_frame
    if(ret):
        # compute the projection and render the cube
        render_frame = renderCube(current_frame,new_intrinsics,R,T)

    # display the current image frame
    cv2.imshow('frame', render_frame)
    k = cv2.waitKey(1)
    if k == 27 or k==113:  #27, 113 are ascii for escape and q respectively
        #exit
        break
```

Then add the top of your code add the following:

```
# This function is yours to complete
# it should take in a set of 3d points and the intrinsic matrix
# rotation matrix(R) and translation vector(T) of a camera
# it should return the 2d projection of the 3d points onto the camera defined
# by the input parameters
def ProjectPoints(points3d, new_intrinsics, R, T):

    # your code here!

    return points2d

# This function will render a cube on an image whose camera is defined
# by the input intrinsics matrix, rotation matrix(R), and translation vector(T)
def renderCube(img_in, new_intrinsics, R, T):
    # Setup output image
```



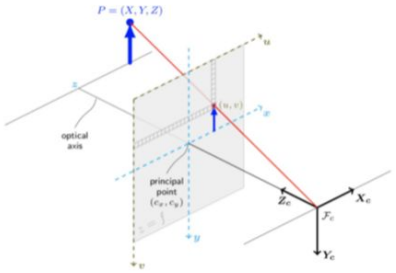
```
    img = np.copy(img_in)

    # We can define a 10cm cube by 4 sets of 3d points
    # these points are in the reference coordinate frame
    scale = 0.1
    face1 = np.array([[0,0,0],[0,0,scale],[0,scale,scale],[0,scale,0]],
                     np.float32)
    face2 = np.array([[0,0,0],[0,scale,0],[scale,scale,0],[scale,0,0]],
                     np.float32)
    face3 = np.array([[0,0,scale],[0,scale,scale],[scale,scale,scale],
                      [scale,0,scale]],np.float32)
    face4 = np.array([[scale,0,0],[scale,0,scale],[scale,scale,scale],
                      [scale,scale,0]],np.float32)
    # using the function you write above we will get the 2d projected
    # position of these points
    face1_proj = ProjectPoints(face1, new_intrinsics, R, T)
    # this function simply draws a line connecting the 4 points
    img = cv2.polylines(img, [np.int32(face1_proj)], True,
                             tuple([255,0,0]), 3, cv2.LINE_AA)
    # repeat for the remaining faces
    face2_proj = ProjectPoints(face2, new_intrinsics, R, T)
    img = cv2.polylines(img, [np.int32(face2_proj)], True,
                             tuple([0,255,0]), 3, cv2.LINE_AA)

    face3_proj = ProjectPoints(face3, new_intrinsics, R, T)
    img = cv2.polylines(img, [np.int32(face3_proj)], True,
                             tuple([0,0,255]), 3, cv2.LINE_AA)

    face4_proj = ProjectPoints(face4, new_intrinsics, R, T)
    img = cv2.polylines(img, [np.int32(face4_proj)], True,
                             tuple([125,125,0]), 3, cv2.LINE_AA)
    return img
```

As you can see, you will need to implement the ProjectPoints function yourself based on what we learned in lecture. Remember that the projection is given by:

points_3D R, T, intrinsic mat

[x, y, z]

points_2D



$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$

$$x' = x/z$$
$$y' = y/z$$
$$u = f_x * x' + c_x$$
$$v = f_y * y' + c_y$$

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

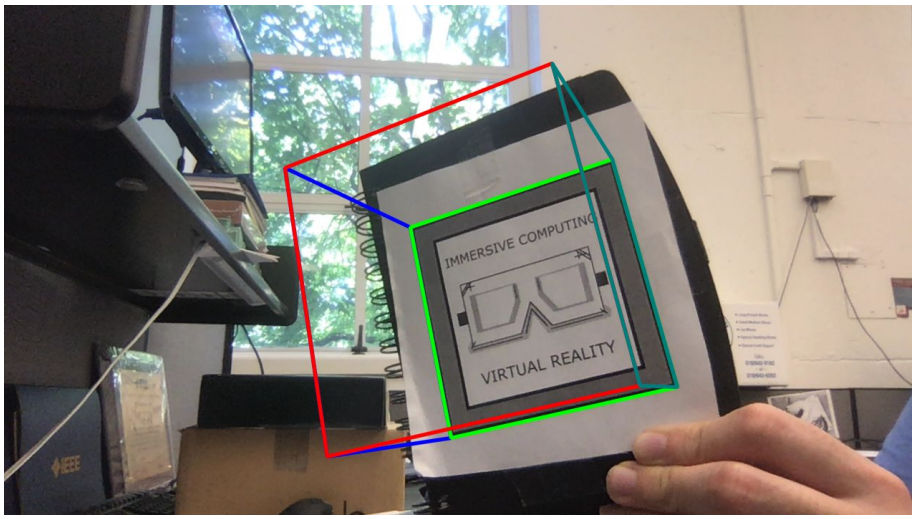
Intrinsic parameters | Extrinsic parameters

where:

- (X, Y, Z) are the coordinates of a 3D point in the world coordinate space
- (u, v) are the coordinates of the projection point in pixels
- A is a camera matrix, or a matrix of intrinsic parameters
- (cx, cy) is a principal point that is usually at the image center
- fx, fy are the focal lengths expressed in pixel units.

If you have correctly implemented the ProjectPoints function, when you hold up your printed reference image, there should be a wireframe cube overlaid on the image. Note: you may have to move the image around a bit to find a position where enough correct feature matches can be computed.

Example:



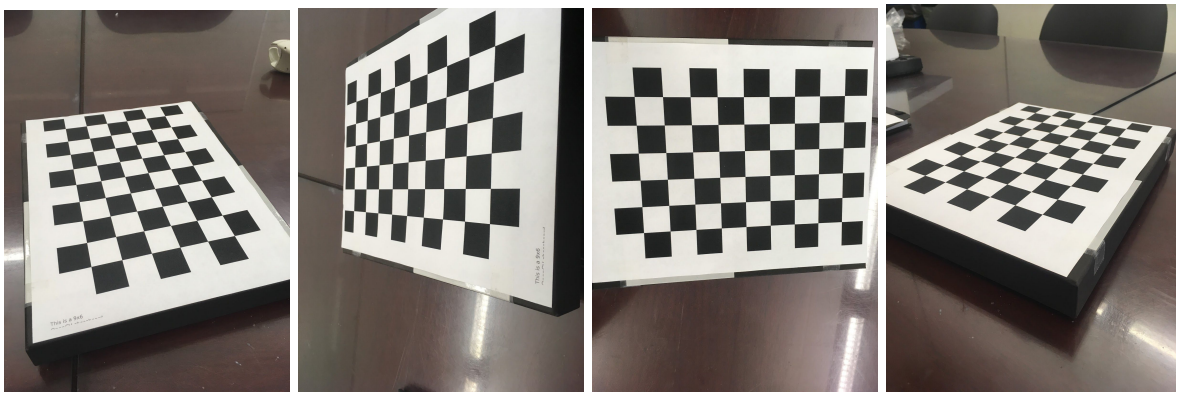
Congrats! You have implemented an AR image tracker. Make sure you include footage of your tracker running in your video.

Part 3 - Mobile Phone 3D Reconstruction:

In this section you will be using what you have learned in the previous section as well as what you learned in lecture to compute a sparse 3d reconstruction of a scene using images collected from your mobile device. We will guide you through the steps to do this, but any new code will be written entirely by you.

Step 1: Calibrate your mobile device camera

As you did with your webcam, you will collect a set of images of the printed checkerboard. This time, you can simply leave the checkerboard on a desk/table and move your camera to collect the images. Remember that you will want roughly 20 images. Examples:

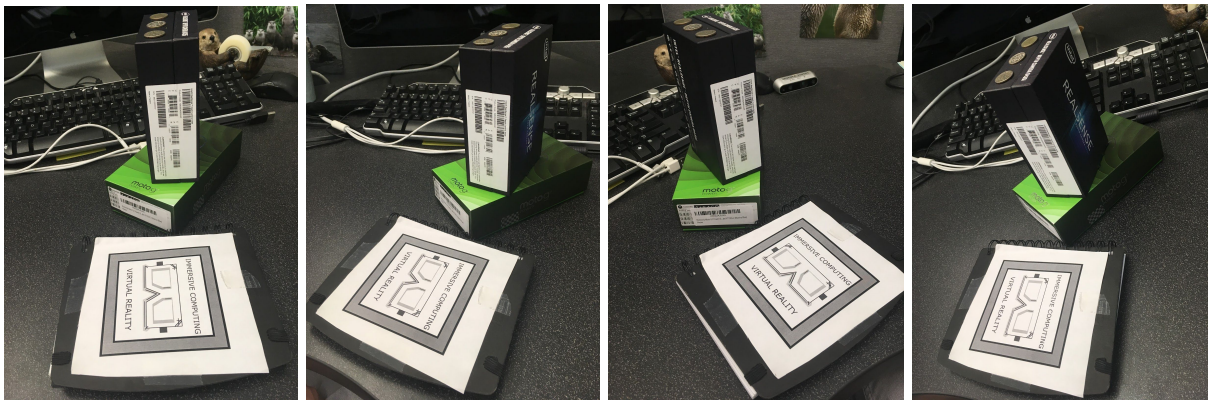


Some notes: You will want to make sure all your photos are oriented the same way (either horizontally or vertically). Going forward you will likely want to downscale your image (using the cv.resize function). You will need to do this both before camera calibration and in your pose estimation code as resizing an image changes its intrinsic calibration matrix.

Step 2: Collect a set of images of the reference image next to an object in the scene

Place your printed reference image next to an object you would like to have a reconstruction of. Your object should have many visual features and an easily recognizable shape (a box with lots of visual texture is a great option). Remember the distance/angle you needed to hold your printed reference at to get good tracking. You should have at least 5 images where you are able to get a good pose estimate.

Examples:



Step 3: Use the AR image tracker code compute the pose of the camera with respect to the printed reference image

Copy your AR image tracker code to a new file, and modify it to load the images from a directory, rather than from the webcam. Compute the pose of each image. If not enough images have a successful pose computation, return to Step 2 and collect additional images. You should try rendering the cube on these images to ensure your pose computation is working correctly. You can delete the images whose pose computation failed as this will simplify computation going forward.

Step 4: Compute relative pose between cameras

In Step 3, you computed the pose of each camera with respect to the reference image. Now you need to compute the relative pose between cameras. In the following steps you are going to compute the 3d position of all features present in the **first** camera image (which we will call Image1), thus we would like to find the pose of each subsequent image relative to that of Image1.

If R_{1r} and T_{1r} represent the pose of the first camera image relative to the reference image, and R_{2r} and T_{2r} represent the pose of the second camera image relative to the reference image.

The rotation from image 1 to image 2 is given by: $R_{21} = R_{2r} * R_{1r}^{T}$
The translation from image 1 to image 2 is given by: $T_{21} = T_{2r} - R_{2r} * R_{1r}^{T} * T_{1r}$

Do this to compute R_{31}, T_{31}, R_{41}, T_{41}, …

Note: Matrix multiplication using numpy is NOT the * operator. Use np.matmul(A,B) to multiply the matrices A and B.

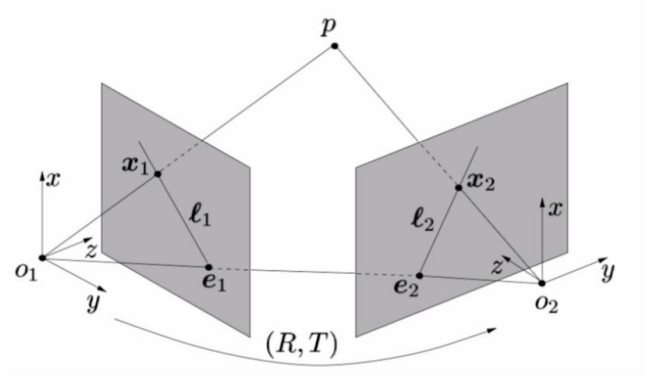
Step 5: Compute feature matches between images

In the previous section we computed matches between features from a reference image and the current webcam image. Now, we also want to compute feature matches between images. This will allow us to compute the 3d position of features, including those outside our reference image. To simplify things, you should compute feature matches between the first camera image and each subsequent image. That is:
Image 1 <-> Image 2
Image 1 <-> Image 3
Image 1 <-> Image 4
…

Example:

Step 6: Use epipolar constraints to remove bad feature matches

Remember that all feature matches must satisfy:



$$
\begin{aligned}
& X_2 = RX_1 + T \\
\Rightarrow\ & \lambda_2 \mathbf{x}_2 = \lambda_1 R\mathbf{x}_1 + T \\
\Rightarrow\ & \lambda_2 \hat{T}\mathbf{x}_2 = \lambda_2 \hat{T} R\mathbf{x}_1 \\
\Rightarrow\ & \mathbf{x}_2^T \hat{T}\mathbf{x}_2 = 0 = \mathbf{x}_2^T \hat{T} R\mathbf{x}_1
\end{aligned}
$$

Or more concisely: $\mathbf{x}_2^T E \mathbf{x}_1 = 0$

Where E is the essential matrix defined as: $E \doteq \hat{T}R$

Note: R and T here are R_{21} and T_{21} calculated above.

And $\mathbf{x}_1$ and $\mathbf{x}_2$ are the feature positions in image 1 and image 2 respectively such that $\mathbf{x}$ = [x, y, 1] = [(u-cx)/fx, (v-cy)/fy, 1].

Note: Remember that the “^” above T indicates the cross product. The cross product in numpy is np.cross but it’s axis may not be correct by default. Use np.cross(A,B,aaxis=0,baxis=0) to compute the cross product between A and B.

Using the above we can check each feature match to see if it satisfies this constraint. In practice, due to noise and numerical issues this may not be exactly zero and so we should check if it is below a threshold.

Write a function that takes in: the camera intrinsic matrix, a set of matches between two images, a set of points in each image, and a rotation and translation between the images, and a threshold parameter. The function should return an array of either 0 or 1 for each point, where 1 represents an inlier and 0 an outlier (outlier = incorrect match).

That is:

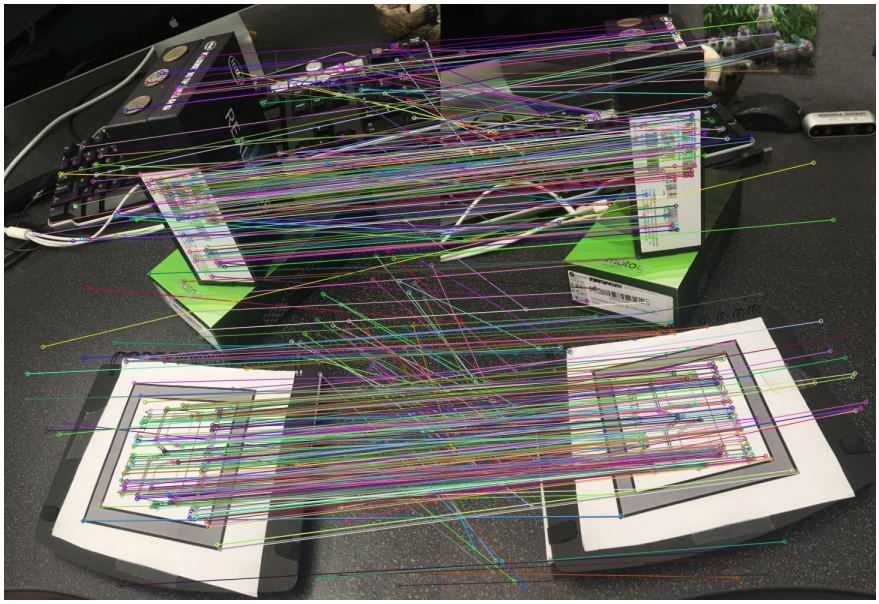
```
def FilterByEpipolarConstraint(intrinsics, matches, points1, points2, Rx1, Tx1,
                               threshold = 0.01):
    # your code here
    return inlier_mask
```

If you then visualize the matches with the drawMatches function:

```
match_visualization = cv2.drawMatches(
    image1, image1_keypoints,
    current_frame,
    current_keypoints, matches, 0,
    matchesMask =inlier_mask, #this applies your inlier filter
    flags=cv2.DRAW_MATCHES_FLAGS_NOT_DRAW_SINGLE_POINTS)
```

You should see the incorrect matches are now mostly (but probably not entirely) removed. You may need to play with the threshold parameter a bit to get a good result.

Example:



Unfortunately satisfying the Epipolar Constraint is a necessary but not sufficient condition for determining correct matches and so it is still possible to get incorrect matches after applying this filter (notice the yellow line in the middle of the above image image). That said, this will be good enough for this homework going forward.

Step 7: Compute feature tracks

Unfortunately there are many more features detected than can be matched properly in every image. Further, multiple observations are what allow us to get accurate depth estimates for each feature. Thus going forward we would like to filter our features such that we only keep the features that can be matched between at least four images. (Image 1 and three others).

To do this you will build what are typically called feature tracks. These are simply a “track” of which images a feature is detected in, and it’s position in these images. Really this is simply a data structure problem that allows us to easily remove features that do not have enough observations.

Compute feature tracks and remove the features that do not have enough observations.

Step 8: Compute the depth for all features with at least three observations

I am including below the relevant slide from lecture notes for reference.

$$\lambda_2^j \boldsymbol{x}_2^j = \lambda_1^j R \boldsymbol{x}_1^j + \gamma T, \quad j = 1, 2, \ldots, n. \tag{5.17}$$

Notice that since (R, T) are known, the equations given by (5.17) are linear in both the structural scale λ's and the motion scale γ's, and therefore they can be easily solved. For each point, λ_1, λ_2 are its depths with respect to the first and second camera frames, respectively. One of them is therefore redundant; for instance, if λ_1 is known, λ_2 is simply a function of (R, T). Hence we can eliminate, say, λ_2 from the above equation by multiplying both sides by $\widehat{\boldsymbol{x}_2}$, which yields

$$\lambda_1^j \widehat{\boldsymbol{x}_2^j} R \boldsymbol{x}_1^j + \gamma \widehat{\boldsymbol{x}_2^j} T = 0, \quad j = 1, 2, \ldots, n. \tag{5.18}$$

This is equivalent to solving the linear equation

$$M^j \bar{\lambda}^j \doteq \begin{bmatrix} \widehat{\boldsymbol{x}_2^j} R \boldsymbol{x}_1^j, & \widehat{\boldsymbol{x}_2^j} T \end{bmatrix} \begin{bmatrix} \lambda_1^j \\ \gamma \end{bmatrix} = 0, \tag{5.19}$$

Notice that all the n equations above share the same γ; we define a vector $\vec{\lambda} = [\lambda_1^1, \lambda_1^2, \ldots, \lambda_1^n, \gamma]^T \in \mathbb{R}^{n+1}$ and a matrix $M \in \mathbb{R}^{3n \times (n+1)}$ as

$$M \doteq \begin{bmatrix} \widehat{\boldsymbol{x}_2^1} R \boldsymbol{x}_1^1 & 0 & 0 & 0 & 0 & \widehat{\boldsymbol{x}_2^1} T \\ 0 & \widehat{\boldsymbol{x}_2^2} R \boldsymbol{x}_1^2 & 0 & 0 & 0 & \widehat{\boldsymbol{x}_2^2} T \\ 0 & 0 & \ddots & 0 & 0 & \vdots \\ 0 & 0 & 0 & \widehat{\boldsymbol{x}_2^{n-1}} R \boldsymbol{x}_1^{n-1} & 0 & \widehat{\boldsymbol{x}_2^{n-1}} T \\ 0 & 0 & 0 & 0 & \widehat{\boldsymbol{x}_2^n} R \boldsymbol{x}_1^n & \widehat{\boldsymbol{x}_2^n} T \end{bmatrix}. \tag{5.20}$$

Then the equation

$$M \vec{\lambda} = 0 \tag{5.21}$$

The above describes the course we will take in computing the depth of each tracked point. The vector of λ's describes the depth of all the tracked features in Image1 with γ representing the overall scale of the scene. Each column in the matrix M represents the constraints on the depth of a feature in Image1. Each row block in the matrix M represents an observation of a feature from Image1 in another image. (We use "row block" instead of "row" here because each $\widehat{\boldsymbol{x}_2^1} R \boldsymbol{x}_1^1$ and $\widehat{\boldsymbol{x}_2^1} T$ is actually a 3x1 vector)

Given the above we know that if we had exact feature measurements we could find a vector of λ's such that: $M \vec{\lambda} = 0$. However, due to noise in the feature measurements we are unlikely to find an exact solution. Instead, what we would like to find is a vector of

λ's that minimizes $M\vec{\lambda}$. To do this we can use SVD!

Specifically (to quote wikipedia):
"**Total least squares minimization:**
A total least squares problem refers to determining the vector x which minimizes the 2-norm of a vector Ax under the constraint ||x|| = 1. The solution turns out to be the right-singular vector of A corresponding to the smallest singular value."

This means that the right-singular vector of M corresponding to the smallest singular value will represent the normalized depth vector of the scene. To get the true depth of all the features in the scene (since in this case we know the true scale of each translation), we can simply divide the vector by γ (the last element of the vector).

In code this simply looks like:

```
W,U,Vt = cv2.SVDecomp(M)
depths = Vt[-1,:]/Vt[-1,-1]
```

So your job for this section is to build the matrix M.

Each row will represent one measurement of a feature in image1 in a subsequent image with the rotation constraint placed in the column for the feature you are observing.

Step 9: Compute the 3d position of the tracked features in the image

This step is very simple, for each feature: $X_1 = \lambda \mathbf{x_1}$
Where X_1 is the 3d position of the feature and λ is the depth of that feature.
Create a Nx3 numpy array of all your 3d feature positions. (Where N is the number of features)

Step 10: Visualize the sparse point cloud

We will use Open3d to visualize the point cloud. First install Open3d
pip install open3d

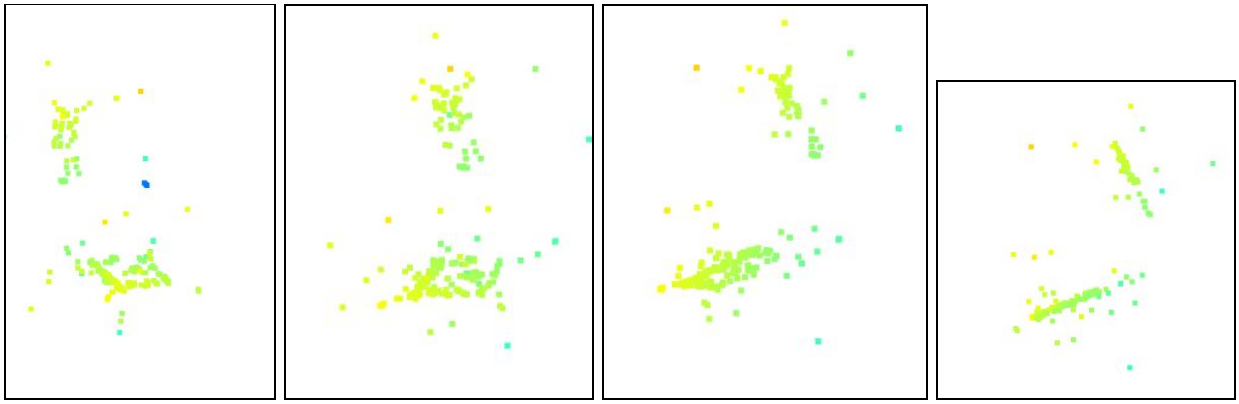
Then to visualize your point cloud simply run:

```
import open3d as o3d
```

```
pcd = o3d.geometry.PointCloud()
pcd.points = o3d.utility.Vector3dVector(your_pointCloud)
o3d.visualization.draw_geometries([pcd])
```

Where “your_pointCloud” is the Nx3 numpy array created in Step 9

Example result:



These are multiple views of the same set of points as it is hard to see what is going on from only one view. The two clusters of points are the box(top) and the printed reference image(bottom).