# Berkeley Deep Drive Camera Calibration

Ajay Ramesh
University of California, Berkeley
Cory Hall, Berkeley, CA 94720, USA
ajayramesh@berkeley.edu

## Abstract

*This paper outlines the current methods we are using for computing the intrinsic and extrinsic parameters of the cameras, as well as the relative pose between two cameras. Our methods involve using collections of coplanar AprilTag markers dubbed "AprilBoards" for estimating the pose of each camera. Additionally, we have built a camera calibration toolkit for internal use, containing several convenience methods for computing intrinsic and extrinsic parameters using footage containing AprilBoards.*

## 1. Introduction

Camera calibration is used for and not limited to 3D reconstruction, depth estimation through stereo vision, visual odometry, and camera-lidar synchronization, all of which are critical to achieving autonomous driving. A calibrated camera often has two components, its intrinsic parameters and its extrinsic parameters. The intrinsic parameters contain focal lengths, the principle point, and distortion coefficients[1]. Extrinsic parameters encode the pose of the camera relative to the object it's taking a picture of. Together, they model the projection of a 3D world point, onto the 2D image plane. Obtaining these parameters allow us to learn more about the physical environment the camera is in when taking a picture.

## 2. AprilTag Cython Wrapper

AprilTag is a type of square fiducial marker in the vain of QR codes and ARToolkit. We use it to relate information about the real world such as distances and lens distortion to the data captured on the camera image plane. We chose to use AprilTag because we only need to store small payloads in the markers, and the detection algorithm is robust in terms of reporting corners accurately and detecting and decoding markers that are far away from the camera. The current implementation of the AprilTag detection algorithm is in C. Much of our infrastructure is written in Python in order to promote rapid prototyping and to take advantage of the several existing software packages available through *pip* and *Anaconda*. In order to seamlessly integrate AprilTag into our existing software stack, we set out to build a Cython wrapper for the C implementation of AprilTag.

### 2.1. Brief Cython Overview

Cython is its own programming language compiled using the Cython compiler. Its strengths include the interoperability of Python and C, declaring C types on Python variables, and calling C functions. Cython code gets compiled into one or more C files which can then be compiled into an archive of shared object binary. We choose to use shared object binaries so that the wrapper can coexist with the original AprilTag library without taking up extra space. Aside from wrapping C code like we are doing, Cython is commonly used for writing highly performant Python code.

### 2.2. Detecting AprilTags in Python

The wrapper accepts a single channel OpenCV-Python frame (a flattened 2D numpy array of type uint8), and passes it onto the scanning function found in the AprilTag API. Our wrapper then serializes the returned C struct containing the detections into a DetectionPayload which has a count as well as an array of Detections. Each Detection has an ID (the 4-bit to 12-bit payload encoded in the tag) as well as the pixel positions of its four corners and its center. The end result is a friendly interface to the C library, which allows us to forget about the detection aspect of calibration and focus on core research. Here's an example of the wrapper in action.

```
from tag36h11_detector import detect
import cv2

im = cv2.imread("tagsampler.jpg")
gray = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
h, w = gray.shape
data = gray.flatten().tobytes()

detections = detect(w, h, data).detections
```

## 3. Camera Pose Estimation

There are two main parts of camera calibration. The first is obtaining the intrinsic parameters of the camera, which we will touch on later, and the second is obtaining the extrinsic parameters of the camera. The extrinsic parameters of the camera describe the rotation and translation of the camera relative to a point in 3D space. The following is the equation that models the projection of a 3D point in the world onto the image plane of a pinhole camera. We will examine each of the components in detail.

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

In general, this equation describes the projection of $\begin{bmatrix} X & Y & Z & 1 \end{bmatrix}^T$ a 3D homogenous coordinate in the real world, onto $\begin{bmatrix} u & v & 1 \end{bmatrix}^T$ a 2D homogenous coordinate on the image plane of the camera, scaled by some $s$. Now, moving from left to right, let's examine the $4 \times 4$ matrix $\begin{bmatrix} R \mid t \end{bmatrix}$. This matrix contains a $3 \times 3$ rotation matrix in the top right and a translation vector. This is the matrix we are interested in when we are estimating the camera's extrinsic parameters. The rotation describes the camera's rotation relative to the 3D world point, and the translation describes the camera's translation relative to the 3D world point. Finally, let's examine the $3 \times 4$ camera matrix $K$. This matrix encodes intrinsic parameters of the camera. Once found, it can be reused for all pose estimations. $f_x$ and $f_y$ are the vertical and horizontal focal lengths of the camera. The focal length is the distance between the camera lens and the image plane. Generally, $f_x = f_y = f$ because we assume a pinhole camera. $c_x$ and $c_y$ make up the principle point, the point where the optical axis of the camera intersects the image plane. Initially, it seems obvious that the principle point should be the center of the image plane [4]. However, this is often not the case due to manufacturing anomalies of real world cameras. Always make sure to calibrate the intrinsics before attempting to calculate the extrinsics. At this point, it's important to note that we are given the 3D point $\begin{bmatrix} X & Y & Z & 1 \end{bmatrix}^T$ and 2D point $\begin{bmatrix} u & v & 1 \end{bmatrix}^T$ which make up a single 3D-3D point correspondence, as well as $f_x$, $f_y$, $c_x$, and $c_y$ which make up the reusable camera matrix $K$. To further clarify (1) just describes the projection of a single 3D point onto a single 2D point. Often times, we need to deal with $n$ 2D-3D correspondences. For example, consider the 4 corners of a square fiducial marker. In order to estimate the pose of the camera relative to a 3D object made up of $n$ points, we employ the function cv2.solvePnP, an OpenCV-Python function, which takes in $n$ 2D-3D correspondences, the camera matrix computed elsewhere, and an optional guess to resolve ambiguities in the estimation. The OpenCV-Python implementation of solvePnP uses an iterative optimization algorithm called Levenberg-Marquardt to find the $\begin{bmatrix} R \mid t \end{bmatrix}$ that best minimizes the reprojection error.

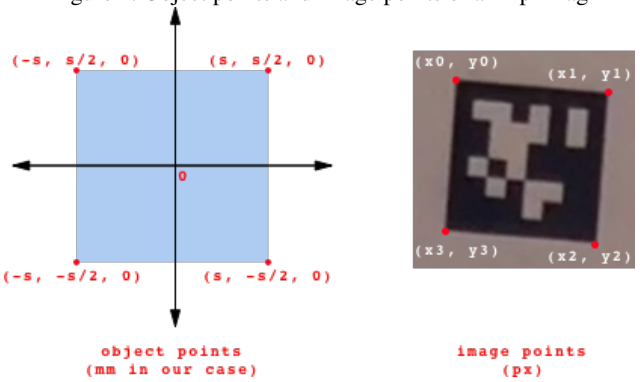
### 3.1. Reprojection Error

The term reprojection error will appear throughout this text, so we will define it here. Let $\mathbf{x}$ be the image point corresponding to some 3D point $\mathbf{v}$. Let $\hat{\mathbf{x}}$ be the reprojected image point. In other words $\hat{\mathbf{x}}$ is the image point we get when passing $\mathbf{v}$ into (1) using some estimated pose $\begin{bmatrix} R \mid t \end{bmatrix}$ while $\mathbf{x}$ is the ground truth. The most optimal $\begin{bmatrix} R \mid t \end{bmatrix}$ will minimize the following error, known as the reprojection error, for two vectors in $\mathbb{R}^n$.

$$E(\mathbf{x}, \hat{\mathbf{x}}) = ||\mathbf{x} - \hat{\mathbf{x}}|| = \sum_{i=1}^{n} |(\mathbf{x}_i - \hat{\mathbf{x}}_i)|^2 \tag{2}$$

### 3.2. Using AprilBoards for Pose Estimation

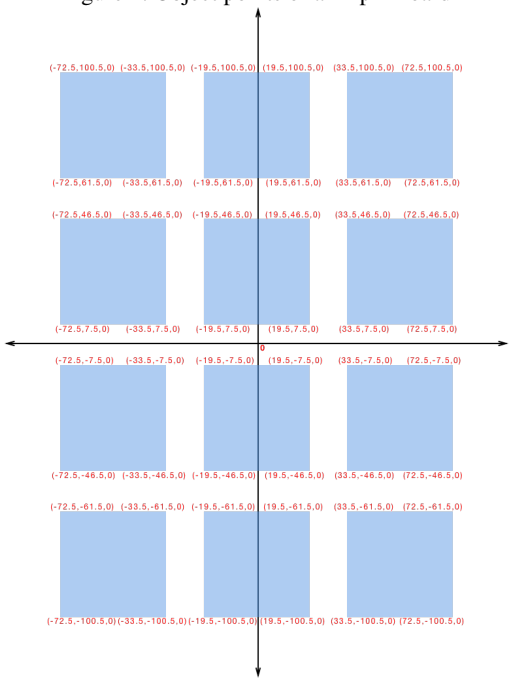
Figure 1. Object points and Image points of an AprilTag




Since we have a way to detect AprilTags in a frame, we can use their corners as image points. The object points are the vertices of a 3D model of the object we have found the image points of. In the case of a single AprilTag, we are dealing with four corners defining a plane. Note that the $Z = 0$ for all the object points. This 3D model can be thought of as an "idealized" AprilTag where it is not transformed in any way. Note that this 3D model has an arbitrary coordinate system imposed upon it where the origin is at the center of the model and vertices are measured relative to it. The object points can be expressed in any units, but we choose to express them in millimeters so that the relative pose can be interpreted in real world units. For example, we will be able to say that the AprilTag is approximately $x$ mm away from the camera. The image points are the corners returned by the AprilTag detection algorithm and are always expressed in pixels. In Figure 1, there are 4 2D-3D correspondences, one of which is $(-s,\ s/2\ ,\ 0)$ and

$(x_0, y_0)$. It turns out that the pose estimate is more accurate when there are more 2D-3D correspondences supplied to the solvePnP algorithm, which is why we came up with the AprilBoard. An AprilBoard is defined as any collection of coplanar AprilTags. If there are $n$ AprilTags on an AprilBoard there are $4n$ point correspondences that can be used for pose estimation. For our first experiment we arbitrarily chose 12 AprilTags which fit comfortably on an A4 sheet of paper, yielding 48 2D-3D point correspondences. All object points are measured in millimeters relative to the origin as seen in Figure 2.

Figure 2. Object points of an AprilBoard




Since each AprilTag has its own unique ID, we can specify which group of IDs are coplanar so that we can get the relative pose between the camera and each board in a scene. In order to sanity check the pose estimation of the camera relative to each board, we can "reproject" the supplied object points back onto the image using the estimated $\left[R \mid t\right]$.

The blue borders around the AprilTags are lines between reprojected image points of each AprilBoard. A good pose estimate yields borders that are close to the actual borders of the AprilTags. In Figure 3, the blue borders fall exactly on the AprilTags on each AprilBoard. The visualization also renders the axis vectors of the coordinate frame of each board, which is oriented the same way as the board. A bad pose estimate yields borders that are far away from the actual borders of the AprilTags. In Figure 4, the first and third board from the left appear distorted. This type of distortion is due to poorly calibrated camera intrinsics. Calibrating camera intrinsics yields distortion coefficients which can correct radial and tangential distortion of the camera lens.

Figure 3. Visualization of a good pose estimate

Figure 4. Visualization of a bad pose estimate

These coefficients are taken into account when computing the relative pose. However, if the coefficients that are passed

in are not representative of the actual distortion, the reprojected image points will not be corrected. This type of distortion can be mitigated by capturing more thorough intrinsic calibration data. The fourth board from the left also does not look correct. This is likely due to an ambiguity between two planar poses [2], or due to poorly reported corners of the AprilTags, which is why the pose is completely off. We can mitigate incorrect pose estimations due to ambiguous poses by providing a "hint" or start state for the Levenberg-Marquardt optimization algorithm. This hint is usually the previous pose estimate, since it is reasonable to assume that the board's position and orientation hasn't changed drastically in a short period of time. The downside to providing a hint is that bad poses can stack up and get worse.

Figure 5. Object point projected onto image plane

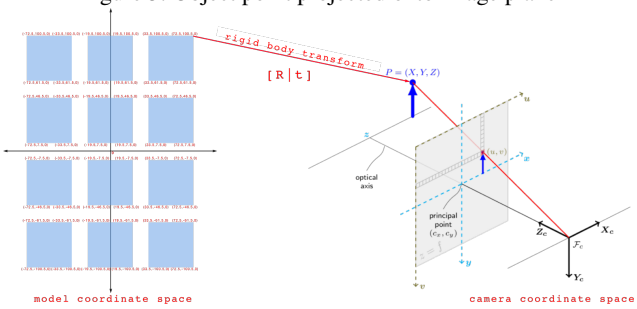



Figure 5 is a visual summary of an object point in model coordinate space being projected onto the image plane of the camera. The pose estimate $[R \mid t]$ rotates and translates the object point from model coordinate space into camera coordinate space. Then, the camera matrix $K$ from (1) projects the 3D point onto the 2D image plane. Another way to visualize and sanity check the pose estimates is to plot the 3D models of the AprilBoards relative to some camera frustum at the origin. Let $\mathbf{x} = \begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix}$ be an object point on an AprilBoard in homogeneous coordinate. Let $[R \mid t]$ be the relative pose of the camera from the AprilBoard.

$$\hat{\mathbf{x}} = [R \mid t]\, \mathbf{x} \tag{3}$$

We can plot $\hat{\mathbf{x}}$, which is a 3D point in the camera coordinate space, relative to the camera frustum in a visualization as shown in Figure 6. This kind of visualization is helpful for identifying incorrect pose estimations. The square pyramid on the far left of the figure is the camera frustum and there are 3 AprilBoards plotted relative to it. The X, Y, and Z axes are measured in millimeters since the object points are measured in millimeters. So, we can say that the boards are about 2 meters away from the camera based on the image alone. Figure 7 shows the 3D visualization of the AprilBoards rendered adjacent to the corresponding frame from the camera.

Figure 6. 3D visualization of AprilBoards

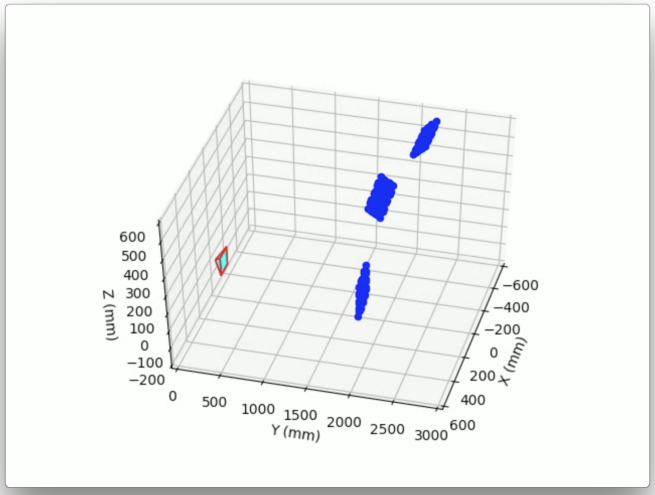
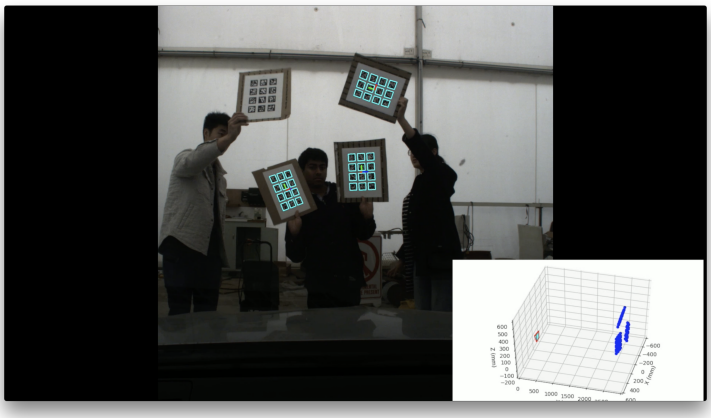



Figure 7. 2D reprojections with 3D visualization



## 4. Calibrating Camera Intrinsics

Thus far we have discussed how we obtain the *extrinsic* parameters of the camera, now we will discuss how we obtain the *intrinsic* parameters of the camera. We have already talked about the camera matrix in equation (1) in Section 3 which takes the optical center, and focal length into account. However, a pinhole camera doesn't have a lens and therefore the pinhole camera model does not correct for lens distortion.

### 4.1. Distortion Coefficients

Radial distortion is easily identified when a picture of an object with straight lines is taken and the resulting image contains the object with curved lines instead. Whereas a fisheye lens deliberately introduces radial distortion, most regular lenses also introduce radial distortion due to the curved nature of the lens. It's responsible for some of the errors we noticed in Figure 4 of section 3. Radial distortion is modeled by the following equations. Let $(x, y)$ be the distorted image point from the input image. The corrected image point after taking radial distortion into account is modeled by the following. [3]

$$x_{\text{corrected}} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \quad (4)$$

$$y_{\text{corrected}} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \quad (5)$$

$$r^2 = x^2 + y^2 \quad (6)$$

Using a known pattern like a chessboard and comparing it against what it looks like on the captured image will allow us to compute the radial distortion coefficients $k_1$, $k_2$, $k_3$ in order to recover the corrected image points. Apart from radial distortion, we must also account for tangential distortion which arises when the image plane is not parallel to the lens. Tangential distortion occurs more rarely so we do not have an example of it in action. Tangential distortion can be modeled by the following.

$$x_{\text{corrected}} = x + [2p_1 xy + p_2(r^2 + 2x^2)] \quad (7)$$

$$y_{\text{corrected}} = y + [p_1(r^2 + 2y^2) + 2p_2 xy] \quad (8)$$

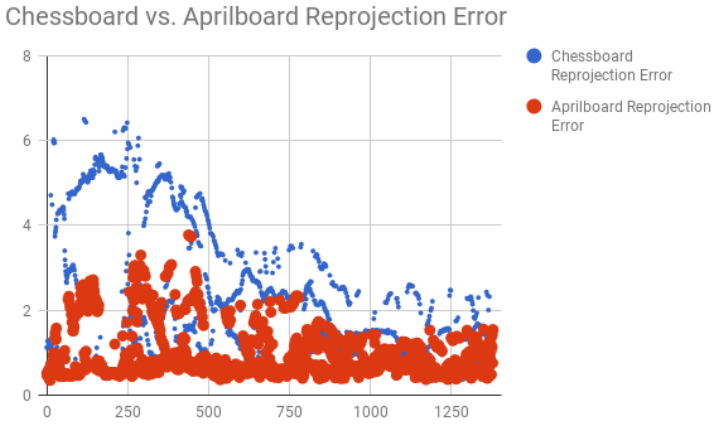
$$r^2 = x^2 + y^2 \quad (9)$$

Therefore, the tangential distortion coefficients are $p_1$ and $p_2$.

### 4.2. Intrinsics from Chessboard vs. AprilBoard

We are using the OpenCV-Python function cv2.calibrateCamera to compute the camera matrix and distortion coefficients. The OpenCV documentation recommends using a chessboard of known size in order to calibrate the camera. While conducting our research we tried using AprilBoards in order to determine the intrinsics and we got much better results. We evaluated both methods by measuring the reprojection error of each frame in a test video, once with the intrinsics from the chessboard, and another time with intrinsics from the AprilBoard. As seen in Figure 8, the average per frame reprojection error of the AprilBoard is much less than the average per frame reprojection error of the Chessboard, over about 1500 frames. The overall average reprojection error for chessboard was 2.3px while the overall average reprojection error for AprilBoard was 0.94px. Another interesting observation is the errors for both methods peak between 0 and 250 frames before getting smaller. This is likely due to bad pose estimations stacking up since we are providing an extrinsic guess. cv2.calibrateCamera computes the intrinsics by minimizing reprojection error by using Levenberg-Marquardt optimization. This involves computing the camera pose. As we established in the previous section, more 2D-3D point correspondences yield more accurate pose estimates. We were using a (4 x 6) chessboard which means the chessboard calibration was using 24 2D-3D point correspondences. The AprilBoard provided 48 2D-3D point correspondences, leading to more accurate pose estimates, and therefore more accurate intrinsics. Our current implementation uses intrinsic parameters computed using AprilBoards.

Figure 8. Chessboard vs AprilBoard Reprojection Error




## 5. Relative Pose Estimation

The relative pose between two cameras is a matrix describing a rotation and translation that takes a point from the coordinate space of one camera, to the coordinate space of another. This transformation allows us to make use of a multiple camera system by enabling depth perception through stereo vision, and allowing us to physically understand the relation among images taken on different cameras on the same time step. We are currently working with two cameras, the left and right cameras. Before computing the relative pose between the two cameras, we must ensure that both cameras take an image at time $t$. If both cameras see the same AprilBoard at time $t$, we can compute the left and right camera's extrinsic parameters relative to this common AprilBoard. Let's call these extrinsics $P_0$ and $P_1$ for the left and right pose estimate respectively. The relative pose can simply be calculated as $P_1 P_0^{-1}$, as shown in Figure 10.

Once again, it helps to write a visualize the relative pose estimates between two cameras. Figure 9 contains the re-

Figure 9. Two camera pose estimation

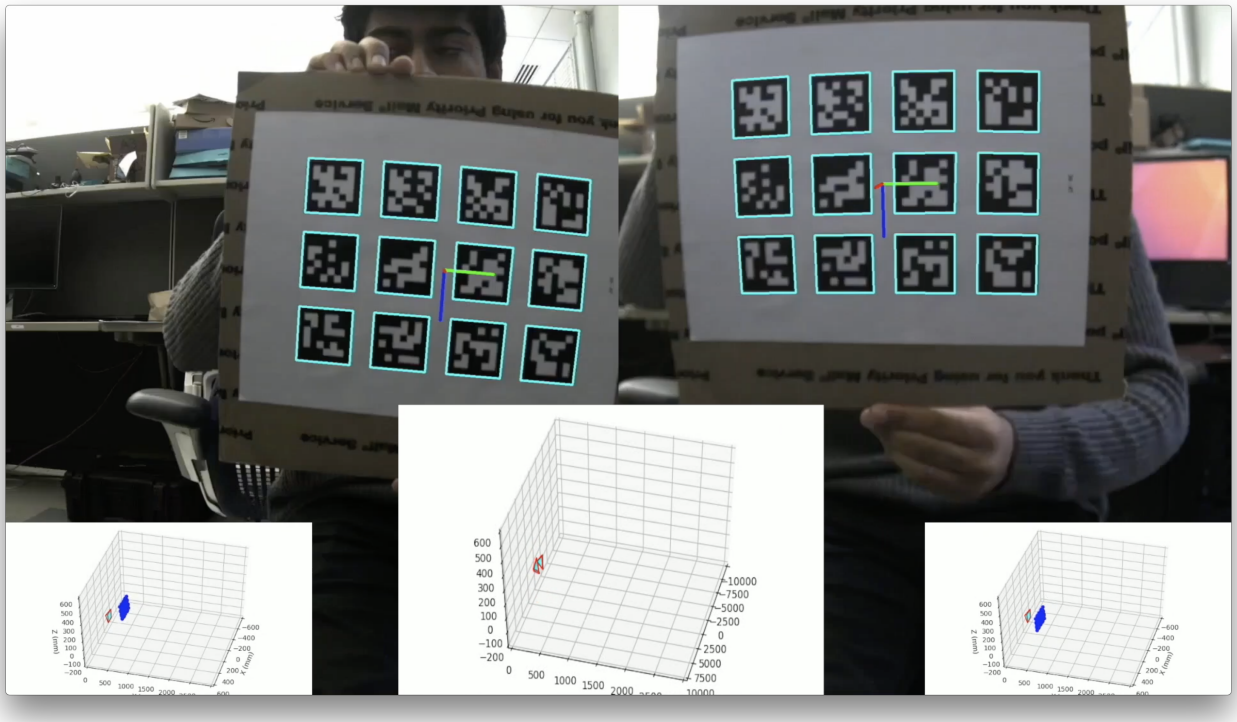
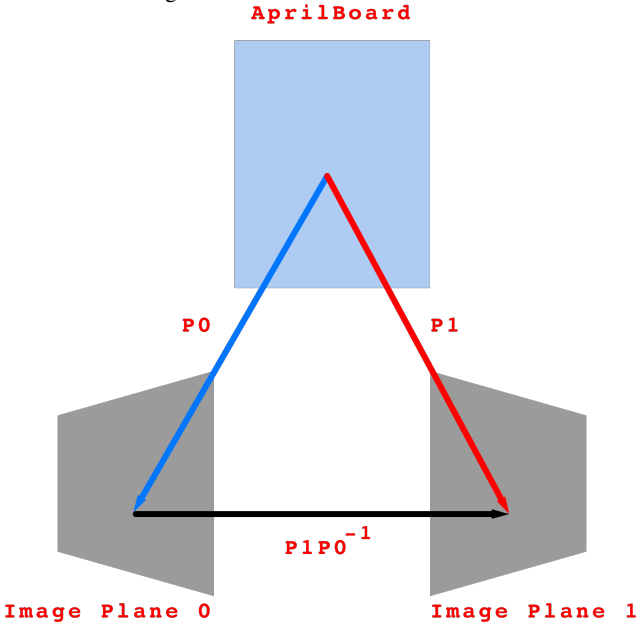


Figure 10. Relative Pose Calculation




projected image points for the left and right cameras. The plot on the far left shows the 3D rendering of the AprilBoard relative to the left camera. The plot on the far right shows the 3D rendering of the AprilBoard relative to the right camera. The plot in the center shows the relative pose between the two camera frustums, in the coordinate space of the left camera. It's clear that this estimation is not perfect. In fact, many of the frames captured in this data set of images yield inaccurate relative pose estimations. We are exploring the following causes for inaccurate relative pose.

- **Synchronization** If the $i$th image taken by the left and right camera was captured at time $t_0$ on the left camera and at time $t_0 + \delta$ on the right camera, the calibration board may have moved slightly in which case the estimation is no longer valid

- **Bad extrinsics** If the extrinsic parameters of the cameras are incorrectly computed, then $P_0$ and $P_1$ are no longer representative of the relative pose between the left and right cameras. As discussed earlier, a bad pose estimate can arise from several different factors including incorrect reporting of corners by the AprilTag detection algorithm as well as ambiguities in the result of solvePnP.

- **Bad intrinsics** If the intrinsic parameters of the camera are incorrectly computed, then the extrinsic parameters will not be able to correct for distortion, leading to an inaccurate $P_0$ and $P_1$. Note that the relative pose estimation in our case relies solely on $P_0$ and $P_1$.

## 6. Camera Calibration Toolkit

All of the above results were computed using a camera calibration toolkit written for internal use in Berkeley Deep Drive. This Python toolkit requires the AprilTag detection wrapper outlined in Section 1. It contains the following features aimed at providing the building blocks for camera calibration without being too opinionated.

- ImageFeed and VideoFeed for abstracting away image sequences into Python iterables
- Scan OpenCV-Python frames for AprilTags
- Get intrinsics from chessboard calibration pattern and store them in a Pickle file
- Get intrinsics from AprilBoard calibration pattern and store them in a Pickle file
- Get camera extrinsics from AprilBoards given a camera feed. Returns several extrinsics depending on which AprilBoards are being searched (determined by bounds of the IDs they contain)
- Render 3D plot of a single camera, and calibration boards relative to it
- Render reprojections of calibration boards from a camera feed, into a video
- Convert Matplotlib figure into an OpenCV-Python frame
- Supports arbitrary AprilBoards defined by the object points and bounds, not just the one we saw in this paper

### 6.1. ImageFeed and VideoFeed

Depending on the data collection method, we may have an input video or a folder of images. Anyone who has worked with OpenCV-Python knows that dealing with images and videos is slightly different in terms of boilerplate required to access the frames. In order to solve this issue, we used the concept of iterables in Python to abstract away image sequences. While writing image and video processing methods it makes sense to supply a frame offset and frame limit to decrease the runtime. Instead of the image processing function having to worry about when to start or stop, we can keep track of the offset and limit in the ImageFeed and VideoFeed.

```
from cct.feeds import ImageFeed, VideoFeed

imfeed = ImageFeed("path/to/images", 20, 100)
vidfeed = VideoFeed("path/to/video", 20, 100)

for frame in vidfeed:
    # stops when the limit is reached
```

## References


[1] R. Hartley and A. Zisserman. Multiple view geometry in computer vision. pages 158–163, 2003. 1

[2] R. Hartley and A. Zisserman. Multiple view geometry in computer vision. page 264, 2003. 4

[3] R. Hartley and A. Zisserman. Multiple view geometry in computer vision. pages 189–191, 2003. 5

[4] J. E. Solem. Programming computer vision with python. pages 109–110, 2012. 2