Independent Study Report (Fall 2022 - Spring 2023)

Jinam Modasiya

June 2023

Introduction

The original intent of the Independent Study in Fall 2022 was to work with a "BlackBox" to record and recreate quadcopter flights indoors. The BlackBox was used to process RF signals generated via a computer Matlab program and send these to a transmitter, which would then relay these messages to a quadcopter. However, this study came short due to equipment malfunction in the BlackBox.

After a few weeks of unsuccessful attempts at troubleshooting, we decided to switch the focus of the Independent Study. Now the Independent Study focused on the use of AprilTags, and their incorporation into object detection, and localization. In Spring 2023, we created recordings of AprilTags attached on quadcopters along with their relative positions, turned into data sets used by Dr. Burlion and Dr. Pompili's lab for further applications.

Fall 2022

Once we switched our focus to object detection and localization using AprilTags, the first few steps involved setup of our main camera and an elementary AprilTag detection algorithm. The camera that was used throughout this entire project is the ArduCam B0205 model. It is capable of recording videos up to 1080P at 30 frames per second. The next step was creating an elementary algorithm that can detect an AprlTag in an image.

0.1 AprilTag Identification

Python has a library called "apriltag", with several functions for working with AprilTags. The code snippets are given in the appendix Code 1.0 and Code 1.1, and explained down below:

The beginning of the Code 1.0 starts off with a variable that captures live video footage from an external camera. The camera records at 30 frames per second at full HD resolution. Each frame is converted into gray-scale and then run through an AprilTag detector function. The output of this detector function is a set of data points which tells us exactly what pixels in the frame correspond to the 4 edges/lines of the AprilTag. This data is then fed into a for loop, which returns the frames with a green line over the pixels that correspond to the edges of the AprilTags. In order to estimate relative positions from the ArduCam's point of view a few parameters are necessary to begin these computations.

0.2 Camera Calibration and Undistortion

While we can find the external dimensions and metrics of the camera from the product website, a calibration is necessary to identify the coefficients for internal parameters of the camera and its lens. These processed were done with the aid of the python library "OpenCV". OpenCV has inbuilt functions which are useful for image processing.

The type of camera used was a pinhole camera. Pinhole cameras introduce significant distortion to images. The two primary distortions are radial distortion and tangential distortion.

Radial distortion causes straight lines to appear curved. As distance from center of the image increases, radial distortion increases.

Radial Distortion can be represented as follows:

$$\begin{aligned} x_{\text{distorted}} &= x\left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right) \\ y_{\text{distorted}} &= y\left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right) \end{aligned}$$

Tangential distortion can occur due to misalignment of the image-taking lens. Due to this misalignment, some areas of the image may look nearer than expected.

The amount of Tangential Distortion can be represented as below:

$$\begin{aligned} x_{\text{distorted}} &= x + \left[2p_1 xy + p_2\left(r^2 + 2x^2\right)\right] \\ y_{\text{distorted}} &= y + \left[p_1\left(r^2 + 2y^2\right) + 2p_2 xy\right] \end{aligned}$$

These distortions can be stacked up and summarized to produce distortion coefficients.

$$\text{Distortion coefficients} = \begin{pmatrix} k_1 & k_2 & p_1 & p_2 & k_3 \end{pmatrix}$$

Beyond the distortion coefficients, intrinsic parameters also need to be estimated. The main two sets of parameters being focal length (f) and optical center (c).

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

The calibration in our case was done using a 9*7 checkerboard with 22 mm side lengths. The code snippet for calibration is given in Appendix Code 2.0.

The results of the calibration are shown in Figure 1.

Undistortion
The main chunk of code to fix distortion caused by the camera lens used is given in Appendix Code 3.0. This was when the Fall semester was coming to an end.

```
Searching for chessboard in frame 700...
ok
Searching for chessboard in frame 720...

Performing calibration...
RMS: 1.3044120468357812
camera matrix:
 [[1.27596980e+03 0.00000000e+00 8.71943095e+02]
 [0.00000000e+00 1.28238391e+03 5.84572815e+02]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
distortion coefficients:  [-0.33912776  0.09010657 -0.00428606  0.01378633  0.06
869174]
```

Figure 1: Camera Calibration Results: Coefficient Matrix

0.3 Vicon Camera Systems

Towards the end of the Fall semester, the focus started to shift to collecting ground truth data that could be used to verify the validity of the results from the AprilTag localization algorithm. In order to get this ground truth data, Vicon Camera Systems were used. These are a type of motion capture technology used to track and analyze objects in 3d space. In the Buehler Flight Lab, the system is set up with 8 individual Vicon cameras, but only 6 of them were functional. The ground truth data here refers to the relative position and orientation data between the AprilTags and the ArduCam. The general way to set up Vicon consists of marker placement, camera calibration, and data analysis. Figure 2 shows what the lab and Vicon system look like. Figure 3 shows what a Vicon tracker looks like. Figure 4 gives an example of what a Vicon system camera looks like.

Figure 2: Vicon System

Figure 3: Vicon Tracker

Figure 4: Vicon Camera Example

Spring 2023

0.4 AprilTags and Drone Setup

The general setup that was used consisted of 4 AprilTags, each belonging to a different family, pasted on to a box, which was then attached to the drone. The reason for using 4 different tags on each side of the drone was to make identification of orientation easier. Since each tag could be potentially connected to a certain yaw orientation, this could later prove useful in calculating relative orientation from the ArduCam. The setup in terms of which side of the cube corresponds with which AprilTag family, is shown in Figure 5, along with an example of what the cubes looked like in real life in Figure 6. Figure 7 is similar to figure 7, but with the larger AprilTags. Figures 8 and 9 are examples of both sized tags on a flying drone.

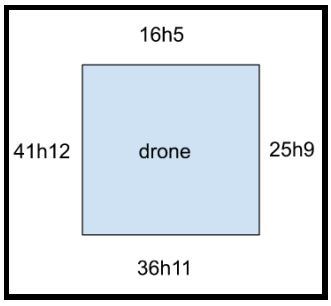


Figure 5: Tag Setup

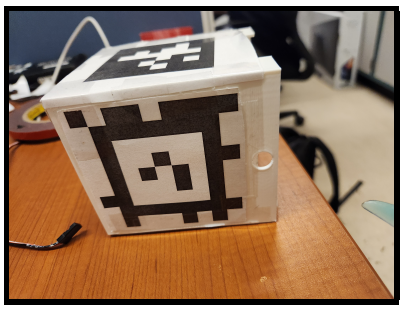


Figure 6: Small Box Example 1

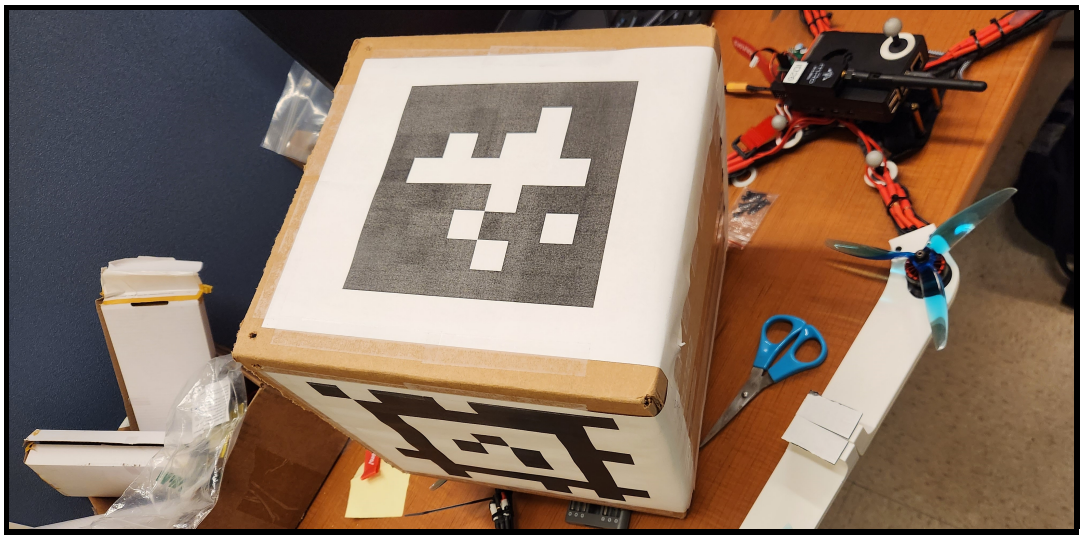


Figure 7: Big Box Example

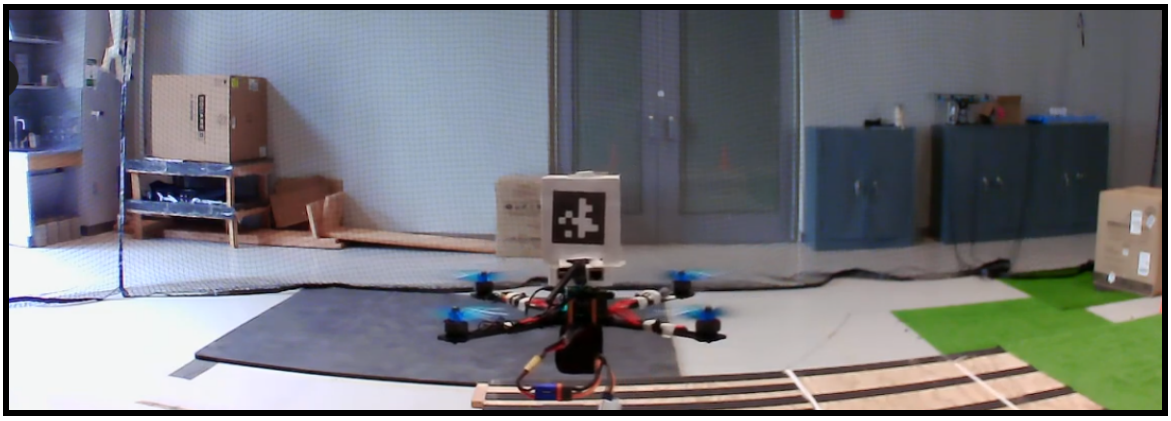


Figure 8: Small Box on Drone Example

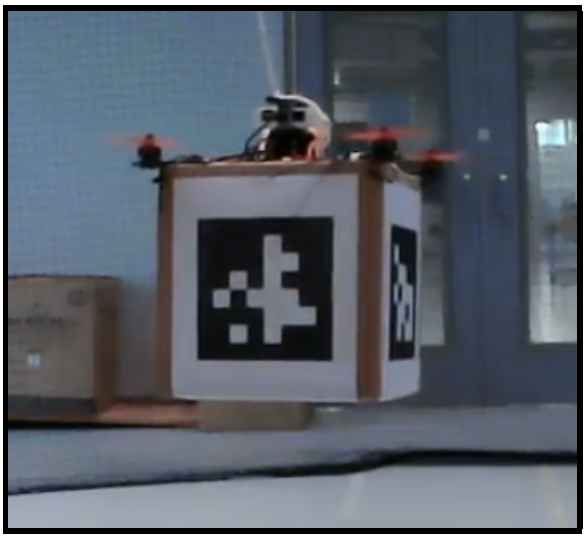


Figure 9: Big Box on Drone

0.5 Vicon Systems Setup

In this case, there were 2 specific objects that needed to be tracked by the Vicon Camera Systems: the AprilTags attached to a drone, and the ArduCam. The ArduCam remained stationary while the drone with AprilTags attached to it was mobile. Trackers were placed on both objects and the data was recorded at 30 frames per second. The picture below shows the point of view of the ArduCam, which is simultaneously recording the drone flight at the same time as the Vicon Camera systems are recording flights.

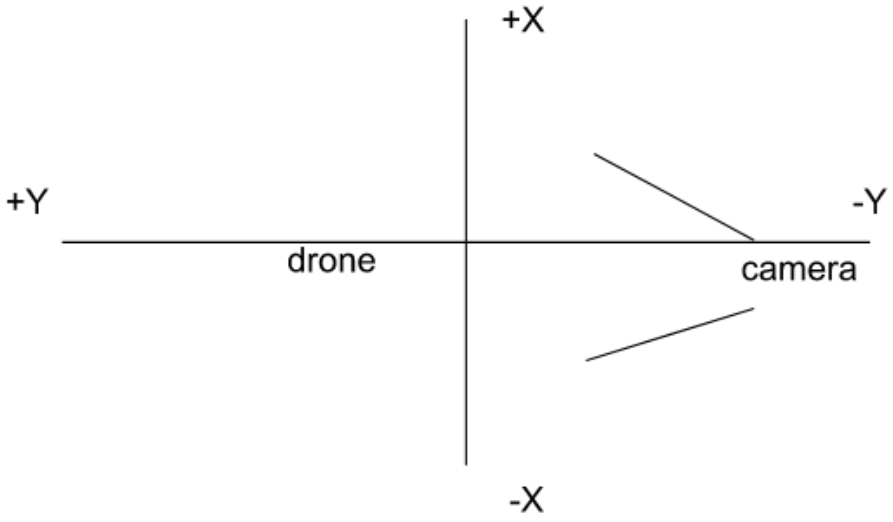
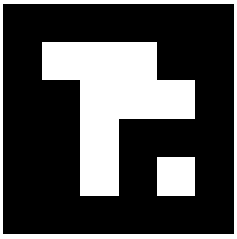


Figure 1: Vicon Setup

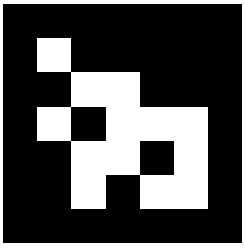
The "Vicon Setup" figure above shows the general setup that was used in relation to the Vicon Camera Systems. The "camera" refers to the ArduCam and the diagonal lines next to it represent the direction in which the field of vision of the ArduCam receives. The "drone" represents the general starting position of the drone. The X and Y axes represent the global set up of the Vicon systems, with Z being height (distance from floor). These can be converted to the relative positions according to the ArduCam camera fairly easily.

On the next page, there are images given of the 4 different AprilTags used, with labels that show which family of tags they belong to.

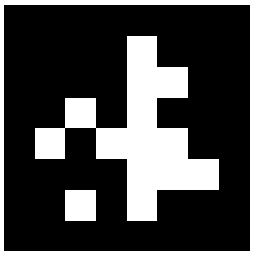
0.6 All 4 AprilTags



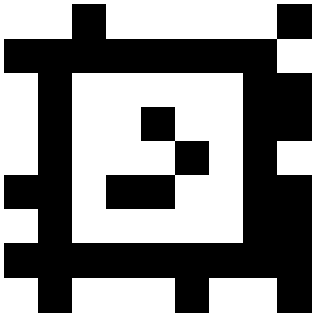
(a) Tag16h5



(b) Tag25h9



(c) Tag36h11



(d) Tag41h12

All 4 of these tags were attached to each visible side of the cubes, and the cubes would be attached to the drone.

0.7 Data using 200 mm AprilTags

Firstly, AprilTags of side length 200 mm were printed and pasted onto a cardboard cube, different families of tags on each side. The initial recordings were done using this setup. Vicon Camera Systems can be operated via a software included with the Vicon systems. The user interface used to manage recordings looks like this:
The "Figure 11: Recording Tab" image shows the tools used for recording data/footage. Under the "System" tab, the capture rate per second was set to 30 Hz, and the cameras were calibrated every time a new recording session had to begin. The field of "Trial Name" would generally be filled with the date and trial number on the particular day, making it easy to identify the recorded files in the future. At the same time, one person would start recording tracking data in the Vicon software, and another person would start recording the video using ArduCam. Once the recording starts, the drone with the tags attached to it is armed, and then a test flight for the desired amount of time commences. After recording, the flights can be replayed virtually using the "LOAD TRIAL" option.

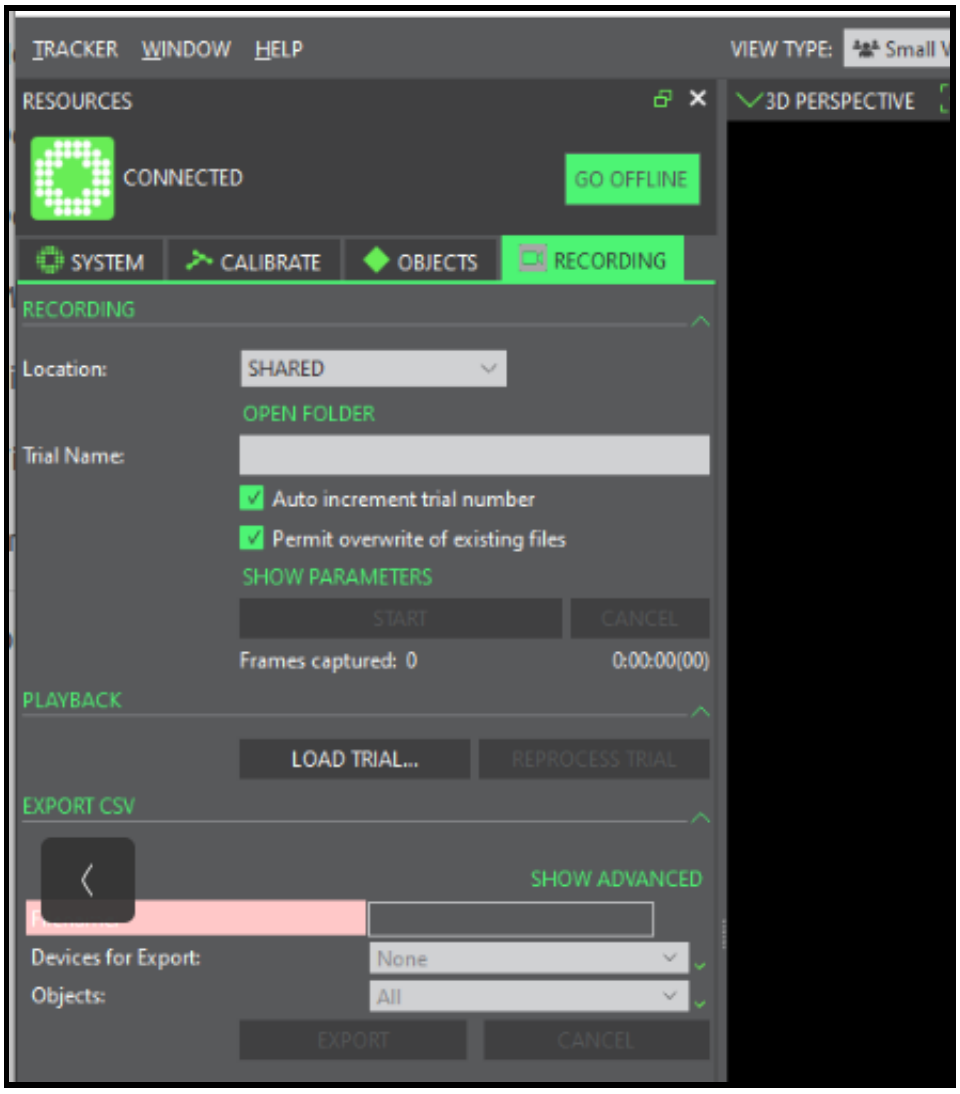


Figure 11: Vicon: Recording Tab

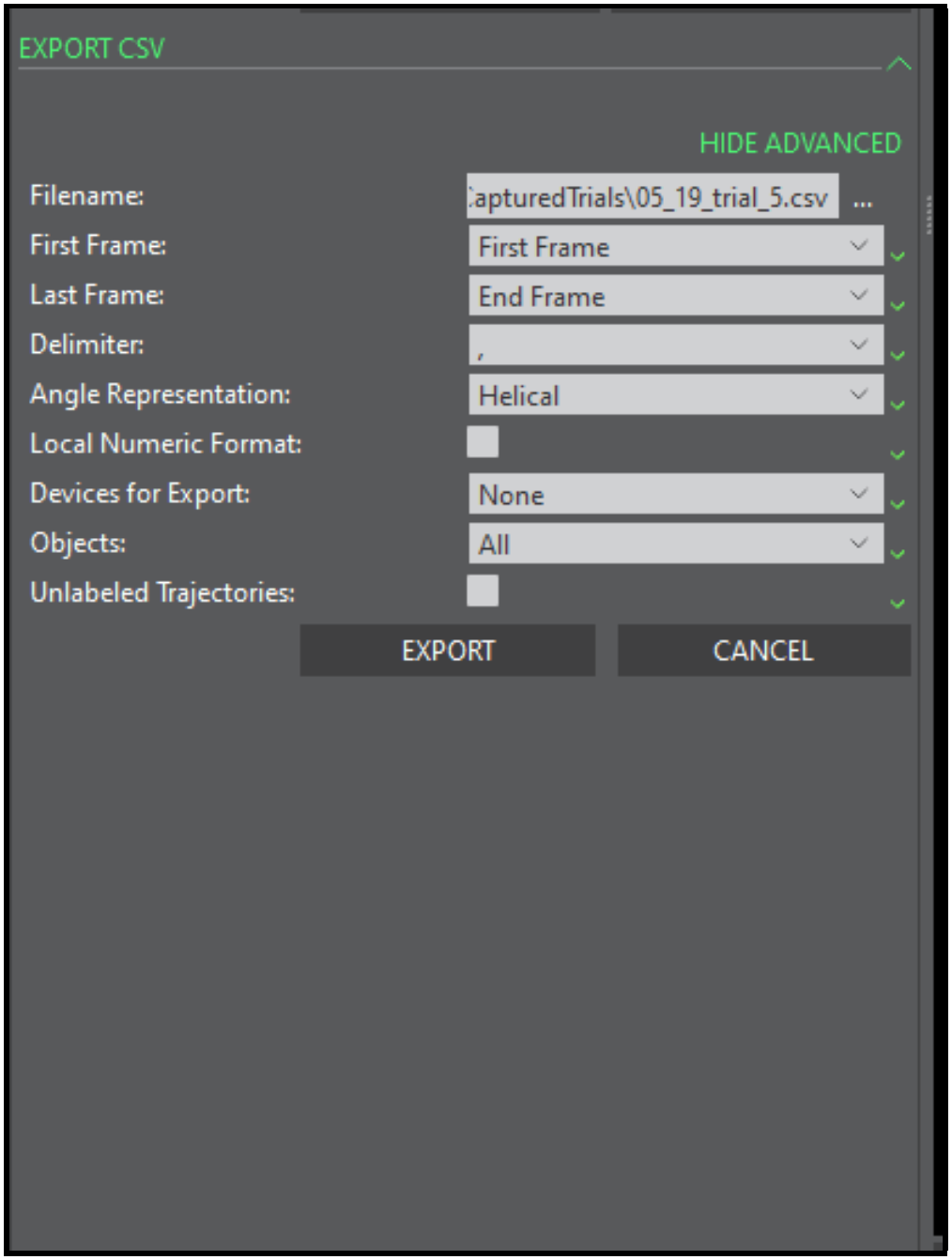


Figure 12: Vicon Export Feature

RX	RY	RZ	TX	TY	TZ
rad	rad	rad	mm	mm	mm
-0.043809	0.048173	-0.143914	-12.531313	201.752777	298.335449
-0.042724	0.048993	-0.14428	-12.49407	201.856293	298.240448
-0.044004	0.048272	-0.143706	-12.514246	201.808395	298.364105
-0.042913	0.048683	-0.144086	-12.520489	201.864487	298.12323
-0.043655	0.048706	-0.143837	-12.514399	201.790802	298.378754
-0.043381	0.048031	-0.143971	-12.521055	201.796875	298.242767
-0.043337	0.049133	-0.143948	-12.479635	201.857819	298.30246
-0.043191	0.048603	-0.143977	-12.560397	201.747009	298.322357
-0.043382	0.047443	-0.143744	-12.486623	201.961029	298.131561
-0.042916	0.049049	-0.144203	-12.519155	201.814911	298.365295
-0.043505	0.048274	-0.14381	-12.529836	201.764145	298.327118
-0.043743	0.0483	-0.144	-12.499794	201.859024	298.225159
-0.043424	0.048978	-0.144053	-12.515001	201.823441	298.324799
-0.043989	0.04774	-0.143968	-12.522699	201.837997	298.223846
-0.043332	0.049124	-0.144127	-12.497328	201.829102	298.314789
-0.043694	0.047999	-0.143884	-12.531171	201.838638	298.284576
-0.043783	0.048383	-0.143768	-12.513674	201.833282	298.322998

Figure 13: Quaternion Angles Example

As we can see in the "Figure 12: Vicon Export Feature" image, after loading a previously recorded trial, it is possible to export out the position data into a CSV format file. The coordinates are according to the virtual Vicon global frame and the angles can be extracted out to be in either helical, or quaternion format. Examples of both quaternion angles, and helical angles are given below:

RX	RY	RZ	TX	TY	TZ
rad	rad	rad	mm	mm	mm
-0.05512	0.068972	-0.056578	-56.16991	274.812897	297.450531
-0.048812	0.069956	-0.053828	-55.81385	275.249176	296.721924
-0.048134	0.070802	-0.05377	-55.854828	275.134308	296.827759
-0.048591	0.070378	-0.053762	-55.874268	275.13266	296.869171
-0.048511	0.07016	-0.05376	-55.880878	275.110291	296.815918
-0.055639	0.068553	-0.056627	-56.203484	274.865265	297.463074
-0.048809	0.069931	-0.053527	-55.902641	275.094238	296.767578
-0.054869	0.068775	-0.056764	-56.141632	274.856232	297.438477
-0.048823	0.070512	-0.053634	-55.880482	275.106018	296.789276
-0.048565	0.07008	-0.053893	-55.802841	275.167511	296.712311
-0.048222	0.071142	-0.053989	-55.827137	275.139923	296.836792
-0.048773	0.070312	-0.053877	-55.844242	275.203766	296.74585
-0.048777	0.069999	-0.053591	-55.881245	275.135468	296.851807
-0.048024	0.071044	-0.054045	-55.835793	275.135529	296.789032
-0.048947	0.069577	-0.053391	-55.87738	275.193024	296.716949
-0.048134	0.070602	-0.054034	-55.877098	275.192627	296.874084
-0.04873	0.07039	-0.05376	-55.879108	275.171387	296.72583

Figure 14: Helical Angles Example

The TX,TY and TZ columns represent relative position in the Vicon global virtual space. The RX, RY, and RZ columns represent relative orientation in the Vicon global virtual space. Figure 13 shows an example of quaternion angles and Figure 14 shows an example of helical angles.

In terms of the actual drone flight, a safety rope was tied around the drone during all the flights because due to the heavy load of the larger box, the drone's flight was unstable and difficult to control.

0.8 Data using 76.2 mm AprilTags

After going over all the data with a 200 mm side length AprilTags, it was decided that something more realistic will be necessary. So, a 4 sided box was created which would house AprilTags of 76.2 mm (3 inches). The process for creating the box started with a basic SolidWorks design as shown in Figure 15. The box was designed with the structure of the drone in mind. The bottom crevaces made it easy to mount the box precisely on top of the drone without causing problems in leveling.

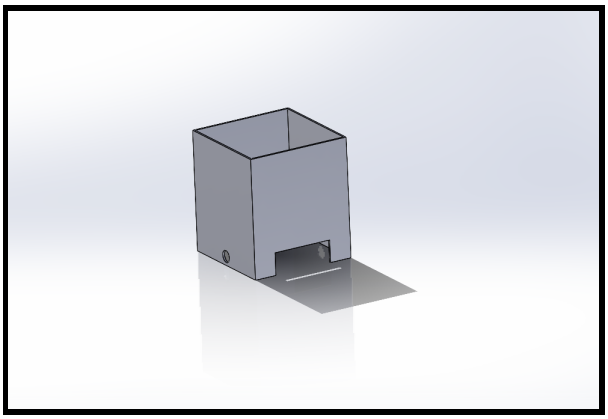


Figure 15: Small Cube Solidworks Model

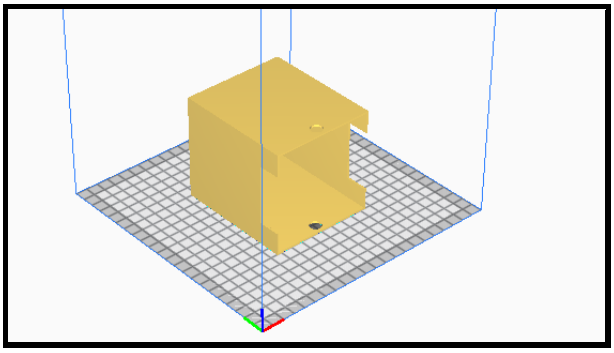


Figure 16: Small Cube Cura Model

Next, Ultimaker Cura was used to convert this SolidWorks file into an STL and then we 3d printed this model. This is what the model looks like in Ultimaker Cura:

With the smaller cube, there was no need to use a safety rope, as the drone was much more stable. The remaining processes to create and record flights and data remained the same, and were repeated with the smaller tags.

All the data is stored at `https://drive.google.com/drive/folders/1v67zo9qByYSwZ0Svb6RVxiiGQ7b0Retb?usp=drive_link`

References

[1] OpenCV Documentation. Camera Calibration. Retrieved from https://docs.opencv.org/4.x/dc/dbb/tutorial$_p y_c alibration.html$

[2] Szeliski, R. (2010). Computer Vision: Algorithms and Applications. Camera Calibration. Retrieved from https://people.cs.rutgers.edu/ elgammal/classes/cs534/lectures/CameraCalibration-book-chapter.pdf

[3] Wikipedia. Distortion (optics). Retrieved from https://en.wikipedia.org/wiki/Distortion

[4] Vicon Documentation. Other hardware documentation. Retrieved from https://docs.vicon.com/display/HD/Other+hardware+documentation

[5] Vicon Documentation. Video cameras documentation. Retrieved from https://docs.vicon.com/display/VideoDoc/Video+cameras+documentation

Appendix

0.9 Code

Code 1.0: AprilTag Detector Part 1

```
cap = cv2.VideoCapture(0)
print("[INFO] loading image...")
while True:

   ret, image = cap.read()
   gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

   print("[INFO] detecting AprilTags...")
   options = apriltag.DetectorOptions(families = "tag36h11")
   detector = apriltag.Detector(options)
   results = detector.detect(gray)
```

Code 1.1: AprilTag Detector Part 2

```
    # loop over the AprilTag detection results
   for r in results:
      # extract the bounding box (x, y)-coordinates for the AprilTag
      # and convert each of the (x, y)-coordinate pairs to integers
      (ptA, ptB, ptC, ptD) = r.corners
      ptB = (int(ptB[0]), int(ptB[1]))
      ptC = (int(ptC[0]), int(ptC[1]))
      ptD = (int(ptD[0]), int(ptD[1]))
      ptA = (int(ptA[0]), int(ptA[1]))
      # draw the bounding box of the AprilTag detection
      cv2.line(image, ptA, ptB, (0, 255, 0), 2)
      cv2.line(image, ptB, ptC, (0, 255, 0), 2)
      cv2.line(image, ptC, ptD, (0, 255, 0), 2)
      cv2.line(image, ptD, ptA, (0, 255, 0), 2)
      # draw the center (x, y)-coordinates of the AprilTag
      (cX, cY) = (int(r.center[0]), int(r.center[1]))
      cv2.circle(image, (cX, cY), 5, (0, 0, 255), -1)
      # draw the tag family on the image
      tagFamily = r.tag_family.decode("utf-8")
      cv2.putText(image, tagFamily, (ptA[0], ptA[1] - 15),
         cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)
      print("[INFO] tag family: {}".format(tagFamily))
   # show the output image after AprilTag detection
   cv2.imshow("Image", image)

   if cv2.waitKey(1) == ord('q'):
      break
cap.release()
cv2.destroyAllWindows()
```

Code 2.0: Camera Calibration

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Calibrate camera using a video of a
        chessboard or a sequence of images.')
    parser.add_argument('input',nargs="?", help='input video file or glob mask')
    parser.add_argument('out',nargs="?",help='output calibration yaml file')
    parser.add_argument('--debug_dir',nargs="?", help='path to directory where images with
        detected chessboard will be written',
                        default='./pictures')
    parser.add_argument('--output_dir',nargs="?",help='path to directory where calibration
        files will be saved.',default='./calibrationFiles')
    parser.add_argument('-c', '--corners',nargs="?", help='output corners file',
        default=None)
    parser.add_argument('-fs', '--framestep',nargs="?", help='use every nth frame in the
        video', default=20, type=int)
    parser.add_argument('--height',nargs="?", help='Height in pixels of the
        image',default=480,type=int)
    parser.add_argument('--width',nargs="?", help='Width in pixels of the
        image',default=640,type=int)
    parser.add_argument('--mm',nargs="?",help='Size in mm of each
        square.',default=22,type=int)
    args = parser.parse_args()

    source = cv2.VideoCapture(0)
    # square_size = float(args.get('--square_size', 1.0))

    pattern_size = (9, 7)
    pattern_points = np.zeros((np.prod(pattern_size), 3), np.float32)
    pattern_points[:, :2] = np.indices(pattern_size).T.reshape(-1, 2)
    # pattern_points *= square_size

    obj_points = []
    img_points = []
    h, w = args.height, args.width
    source.set(cv2.CAP_PROP_FRAME_HEIGHT,h)
    source.set(cv2.CAP_PROP_FRAME_WIDTH,w)

    i = -1
    image_count=0
    image_goal=30
    while True:
        i += 1
        if isinstance(source, list):
            # glob
            if i == len(source):
                break
            img = cv2.imread(source[i])
        else:
            # cv2.VideoCapture
            retval, img = source.read()
```

```
        if not retval:
            break
        if i % args.framestep != 0:
            continue
    cv2.imshow('Image',img)

    key = cv2.waitKey(1) & 0xFF
    if key == ord('q'):
        break
    print('Searching for chessboard in frame ' + str(i) + '...'),
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    h, w = img.shape[:2]
    found, corners = cv2.findChessboardCorners(img, pattern_size,
        flags=cv2.CALIB_CB_FILTER_QUADS)
    if found:
        term = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_COUNT, args.mm, 0.1)
        cv2.cornerSubPix(img, corners, (5, 5), (-1, -1), term)
        image_count=image_count+1
        if image_count==image_goal:
            break
    if args.debug_dir:
        img_chess = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR)
        cv2.drawChessboardCorners(img_chess, pattern_size, corners, found)
        cv2.imwrite(os.path.join(args.debug_dir, '%04d.png' % i), img_chess)
    if not found:
        print ('not found')
        continue
    img_points.append(corners.reshape(1, -1, 2))
    obj_points.append(pattern_points.reshape(1, -1, 3))

    print ('ok')

if args.corners:
    with open(args.corners, 'wb') as fw:
        pickle.dump(img_points, fw)
        pickle.dump(obj_points, fw)
        pickle.dump((w, h), fw)


print('\nPerforming calibration...')
rms, camera_matrix, dist_coefs, rvecs, tvecs = cv2.calibrateCamera(obj_points,
    img_points, (w, h), None, None)
print ("RMS:", rms)
print ("camera matrix:\n", camera_matrix)
print ("distortion coefficients: ", dist_coefs.ravel())

# # fisheye calibration
# rms, camera_matrix, dist_coefs, rvecs, tvecs = cv2.fisheye.calibrate(
#     obj_points, img_points,
#     (w, h), camera_matrix, np.array([0., 0., 0., 0.]),
#     None, None,
#     cv2.fisheye.CALIB_USE_INTRINSIC_GUESS, (3, 1, 1e-6))
```

```
    # print "RMS:", rms
    # print "camera matrix:\n", camera_matrix
    # print "distortion coefficients: ", dist_coefs.ravel()

    calibration = {'rms': rms, 'camera_matrix': camera_matrix.tolist(), 'dist_coefs':
        dist_coefs.tolist() }

    ##OUTPUT DIRECTORIES
    file1 = args.output_dir + "/cameraMatrix.txt"
    np.savetxt(file1,camera_matrix,delimiter=',')
    file2 = args.output_dir + "/cameraDistortion.txt"
    np.savetxt(file2,dist_coefs,delimiter=',')
```

Code 3.0: Image Undistortion

```
if __name__ == '__main__':

    parser = argparse.ArgumentParser(description='Undistort images based on camera
        calibration.')
    parser.add_argument('calibration', help='input video file')
    parser.add_argument('input_mask', help='input mask')
    parser.add_argument('out', help='output directory')
    args = parser.parse_args()

    with open(args.calibration) as fr:
        c = yaml.load(fr)

    for fn in glob(args.input_mask):
        print ('processing %s...' % fn,)
        img = cv2.imread(fn)
        if img is None:
            print("Failed to load " + fn)
            continue

        K_undistort = np.array(c['camera_matrix'])

        img_und = cv2.undistort(img, np.array(c['camera_matrix']),
            np.array(c['dist_coefs']),
                                newCameraMatrix=K_undistort)
        name, ext = os.path.splitext(os.path.basename(fn))
        cv2.imwrite(os.path.join(args.out, name + '_und' + ext), img_und)

        print ('ok')
```