A Solution for April-tag Positioning

Weixu Zhu

December 2017

Abstract

In this report, we present a solution to determine the 3D pose of april-tags. The proposed solution aims specifically to locate a square from the 4 corners of its 2D projection.

1 Introduction

In our work, we need to use a single camera to detect cube-shape blocks, with each of the 6 faces attached an april-tag (shown in Figure 1), and calculate their position and rotation out of the pictures taken by the camera.

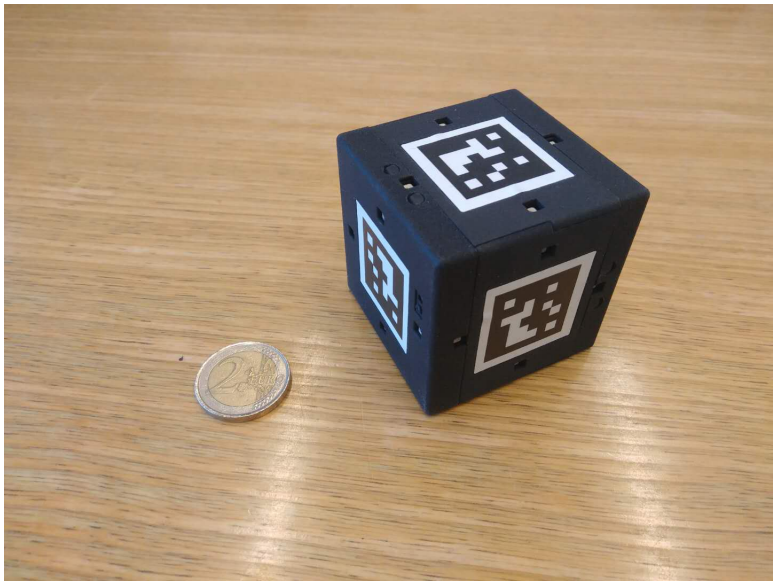

Figure 1: A block

By using April-tag library, we are provided with the position of the center and the 4 corners of each tag by the library[4]. Our goal is to calculate the 3D pose of each tag from these 4 points.

Although April-tag library provides *homography_to_pose* function to achieve this goal, our previous test showing that *solvepnp* function in OpenCV is more stable[1].

However, although this typical p4p problem can be achieved easily by using *solvepnp* function of OpenCV library[3, 2], considering that solvepnp in OpenCV is a generic function for N points, there should be a better solution specifically aiming for positioning 4 points in a square shape.

This report provides the detail of our solution.

2 Square Location Problem

The target is to locate a square in a 3D space from the 2D coordinates of its 4 corners.

For a square, we assume : The centre of the square is (x, y, z). Two vectors are (a, b, c) and (p, q, r), as shown in Figure 2.

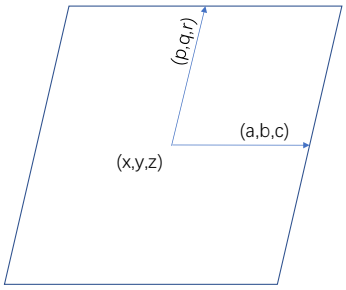

Figure 2: A square in 3D

If the corners of the square are $P_1 : (x_1, y_1, z_1)$, $P_2 : (x_2, y_2, z_2)$, $P_3 : (x_3, y_3, z_3)$ and $P_4 : (x_4, y_4, z_4)$, then we have

$$P_1 : (x_1, y_1, z_1) = (x + a + p, y + b + q, z + c + r) \quad (1)$$

$$P_2 : (x_2, y_2, z_2) = (x + a - p, y + b - q, z + c - r) \quad (2)$$

$$P_3 : (x_3, y_3, z_3) = (x - a - p, y - b - q, z - c - r) \quad (3)$$

$$P_4 : (x_4, y_4, z_4) = (x - a + p, y - b + q, z - c + r) \quad (4)$$

For every frame, we have the coordination of the 4 corners on the picture, (u_1, v_1), (u_2, v_2), (u_3, v_3) and (u_4, v_4). Based on the camera model, we have

$$u_i = k_u \frac{x_i}{z_i} + u_0 \tag{5}$$

$$v_i = k_v \frac{y_i}{z_i} + v_0 \tag{6}$$

where $i = 1, 2, 3, 4$ and k_u, k_v, u_0, v_0 are camera's internal parameters. That is, for every corner P_i, there are

$$-k_u x_i + (u_i - u_0)z = 0 \tag{7}$$

$$-k_v y_i + (v_i - v_0)z = 0 \tag{8}$$

where $i = 1, 2, 3, 4$.

Considering Equation 1, 2, 3 and 4, we can write a linear equation system which consists of 8 equations:

$$\begin{bmatrix} -k_u & & u_1 - u_0 & -k_u & & u_1 - u_0 & -k_u & & u_1 - u_0 \\ & -k_v & v_1 - v_0 & & -k_v & v_1 - v_0 & & -k_v & v_1 - v_0 \\ -k_u & & u_2 - u_0 & -k_u & & u_2 - u_0 & k_u & & -(u_2 - u_0) \\ & -k_v & v_2 - v_0 & & -k_v & v_2 - v_0 & & k_v & -(v_2 - v_0) \\ -k_u & & u_3 - u_0 & k_u & & -(u_3 - u_0) & k_u & & -(u_3 - u_0) \\ & -k_v & v_3 - v_0 & & k_v & -(v_3 - v_0) & & k_v & -(v_3 - v_0) \\ -k_u & & u_4 - u_0 & k_u & & -(u_4 - u_0) & -k_u & & u_4 - u_0 \\ & -k_v & v_4 - v_0 & & k_v & -(v_4 - v_0) & & -k_v & v_4 - v_0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ a \\ b \\ c \\ p \\ q \\ r \end{bmatrix} = \mathbf{0} \tag{9}$$

Other than the linear camera equations (Equation 9), there are two nonlinear constrains indicating that P_1, P_2, P_3 and P_4 make a square, which are

$$ap + bq + cr = 0 \tag{10}$$

and

$$a^2 + b^2 + c^2 = p^2 + q^2 + r^2 \tag{11}$$

and an extra equation constraining the size of the square

$$a^2 + b^2 + c^2 = p^2 + q^2 + r^2 = L^2 \tag{12}$$

where L is half length of the side of the square.

Therefore, the problem becomes: given (u_1, v_1), (u_2, v_2), (u_3, v_3) and (u_4, v_4), and camera internal parameters, we need to solve x, y, z, a, b, c, p, q and r out of Equation 9, 10, 11 and 12

3 Problem Analysis

Since there are more equations than variables, we should be able to exploit an optimization algorithm to solve this problem, but our aim is to make our program be stable, simple and portable, and a slight amount of error is tolerable, so here we propose another solution.

In the linear equation systems (Equation 9), there are 8 equations and 9 variables. If we choose one of the variables from x, y, z, a, b, c, p, q and r as a base variable, for example z, we can get the linear relation of 8 other variables to z:

$$\begin{cases} x = x_z z \\ y = y_z z \\ a = a_z z \\ b = b_z z \\ c = c_z z \\ p = p_z z \\ q = q_z z \\ r = r_z z \end{cases} \tag{13}$$

where x_z, y_z, a_z, b_z, c_z, p_z, q_z and r_z are the linear coefficients which are calculated out of the linear equation system (Equation 9).

Substitute Equation 13 into square size constrain(Equation 12), we get

$$(a_z^2 + b_z^2 + c_z^2)z^2 = L^2 \tag{14}$$

$$(p_z^2 + q_z^2 + r_z^2)z^2 = L^2 \tag{15}$$

From either Equation 14 or Equation 15 we can get the solution of z, and from this z, we can get all the other 8 unknown variables.

However, in the procedure above, we used mainly the linear equation system, which are the camera equations, but we didn't consider any of the nonlinear equations which constrains that P_1, P_2, P_3 and P_4 make a square. The consequence of this could be that although the calculated 3D positions of P_1, P_2, P_3 and P_4 match their projection on the image perfectly, they may not be a square, but a parallelogram.

To fix this, we have to take square constrains(Equation 10 and 11) into consideration, and drop one point from P_1, P_2, P_3 and P_4(two of those linear equations).

If we think about the geometric significance, this makes perfect sense. Each two camera equations for one point of P_1, P_2, P_3 and P_4 means a ray shooting out from the center of the camera, and we need to put a square into these rays making that each corner of the square lies on the rays. Apparently, only three rays are enough to determine the location of this square(there may be multiple solutions, but each one is a fixed pose). If there are four rays, we will have to distort this square into a parallelogram to make it fit in all the 4 rays.

Therefore, here is our final solution:

We use the first 6 equations in the linear equation system, which contains the camera information of 3 corners of the square, and Equation 10, 11 and 12, which contains the information of the square, to solve the 9 unknown variables.

First, we choose 3 variables as the base variables, for example c, r and z. From the 6 linear equations, we can get the linear coefficients of 6 other variables to these 3 base variables, c, r and z:

$$\begin{cases} x = x_c c + x_r r + x_z z \\ y = y_x c + y_r r + y_z z \\ a = a_c c + a_r r + a_z z \\ b = b_c c + b_r r + b_z z \\ p = p_c c + p_r r + p_z z \\ q = q_c c + q_r r + q_z z \end{cases} \tag{16}$$

where x_c, x_r, x_z, y_c, y_r, y_z, a_c, a_r, a_z, b_c, b_r, b_z, p_c, p_r, p_z, q_c, q_r and q_z are the linear coefficients.

Then, we substitute Equation 16 into square constraining equations(Equation 10 and 11). We can get two quadratic equations about c, r and z

$$\begin{cases} A_1 c^2 + B_1 r^2 + C_1 cr + D_1 cz + E_1 rz + F_1 z^2 = 0 \\ A_2 c^2 + B_2 r^2 + C_2 cr + D_2 cz + E_2 rz + F_2 z^2 = 0 \end{cases} \tag{17}$$

where A_1, B_1, C_1, D_1, E_1, F_1, A_2, B_2, C_2, D_2, E_2 and F_2 are known coefficients that can be calculated by the linear coefficients.

Since all the terms in Equation 17 are quadratic terms, we can note

$$\begin{cases} c = c_z z \\ r = r_z z \end{cases} \tag{18}$$

and since z can't be 0, Equation 17 becomes

$$\begin{cases} A_1 c_z^2 + B_1 r_z^2 + C_1 c_z r_z + D_1 c_z + E_1 r_z + F_1 = 0 \\ A_2 c_z^2 + B_2 r_z^2 + C_2 c_z r_z + D_2 c_z + E_2 r_z + F_2 = 0 \end{cases} \tag{19}$$

After solving Equation 19 and getting c_z and r_z, we can substitute Equation 18 into Equation 12 (constraining the size of the square) to solve z.

Therefore, the key problem is to solve the quadratic equation system (Equation 19).

Apparently, to find a algebraic solution for a quadratic equation system is very difficult, so we exploit a little trick to find its numerical solution.

Consider (c_z, r_z) as a point on a 2D plane, Function G_1 and G_2 are functions on this c_z-r_z plane, which are

$$\begin{cases} G_1(c_z, r_z) = A_1 c_z^2 + B_1 r_z^2 + C_1 c_z r_z + D_1 c_z + E_1 r_z + F_1 \\ G_2(c_z, r_z) = A_2 c_z^2 + B_2 r_z^2 + C_2 c_z r_z + D_2 c_z + E_2 r_z + F_2 \end{cases} \tag{20}$$

We can start from an initial point (c_z^0, r_z^0), and evaluate $G_1(c_z^0, r_z^0)$ and $G_2(c_z^0, r_z^0)$, if they are not 0, compensate point (c_z^0, r_z^0) according to the gradients of G_1 and G_2, until $G_1(c_z^0, r_z^0)$ and $G_2(c_z^0, r_z^0)$ are close enough to 0, as shown in Algorithm *solveQuad*.

Theoretically, the quadratic equation system (Equation 19) could have up to 4 solutions. In our strategy we are using gradients to approach the numerical solution, thus it is the initial point (c_z^0, r_z^0) that determines which solution of the 4 we get in the end.

In order to set the initial point close enough to the right solution, we have the 8 camera equations (Equation 9) and their solution (Equation 13) to get a proximate (c_z^p, r_z^p). This proximate (c_z^p, r_z^p) may not satisfy square constrains (Equation 10 and 11), but it serves perfectly as an initial point for the numerical solution of the quadratic equation system (Equation 19), leading us to the right one out of the 4 possible solutions.

---

function SOLVEQUAD(A_1-F_1,A_2-F_2,x_0,y_0,*threshold*)
  $x \leftarrow x_0$, $y \leftarrow y_0$
  while $|G_1(x,y)| > threshold$ or $|G_2(x,y)| > threshold$ **do**
    Gradient:
    $G_{1x} \leftarrow 2A_1x + C_1y + D_1$
    $G_{1y} \leftarrow 2B_1y + C_1x + E_1$
    $G_{2x} \leftarrow 2A_2x + C_2y + D_2$
    $G_{2y} \leftarrow 2B_2y + C_2x + E_2$
    Step:
    $\Delta x_{G1} \leftarrow G_{1x}/\sqrt{G_{1x}^2 + G_{1x}^2}$
    $\Delta y_{G1} \leftarrow G_{1y}/\sqrt{G_{1x}^2 + G_{1x}^2}$
    $\Delta G_1 \leftarrow G_{1x}\Delta x_{G1} + G_{1y}\Delta y_{G1}$
    $\Delta x_{G2} \leftarrow G_{2x}/\sqrt{G_{2x}^2 + G_{2x}^2}$
    $\Delta y_{G2} \leftarrow G_{2y}/\sqrt{G_{2x}^2 + G_{2x}^2}$
    $\Delta G_2 \leftarrow G_{2x}\Delta x_{G2} + G_{2y}\Delta y_{G2}$
    Compensate:
    $x_{new} \leftarrow x + \frac{(0-G_1(x,y))}{\Delta G_1}\Delta x_{G1} + \frac{(0-G_2(x,y))}{\Delta G_2}\Delta x_{G2}$
    $y_{new} \leftarrow y + \frac{(0-G_1(x,y))}{\Delta G_1}\Delta y_{G1} + \frac{(0-G_2(x,y))}{\Delta G_2}\Delta y_{G2}$
    $x \leftarrow x_{new}$, $y \leftarrow y_{new}$
  end while
  return x, y
end function

At last, as soon as we have z, we can have all the other variables, among which x, y, z give the location of the square and a, b, c p, q, r give the rotation of the square. There would be no problem to transform a, b, c p, q, r into other form like rotation matrix or quaternion to indicate the rotation.

4 Experiment and Conclusion

We made a comparison between our result and OpenCV's result. We selected two sets of video taken from our robot which captures frames at 5FPS 640×320, one is a tag just entering the sight, and the other is the robot tracking and

approaching the tag. We used both solvepnp function (we used "CV_P3P" flag, which is Gao's method [3]) from OpenCV and our solution in lua code to calculate the position of that tag appearing in the the video.

By setting up the solvepnp function of OpenCV, we used "CV_P3P" method. In our solution, we set the threshold in solveQuad function as 0.0000000001. The unit of length is in meter.

We compared both the location and the quaternion, the result is shown in Figure 3, 4, 5 and 6.

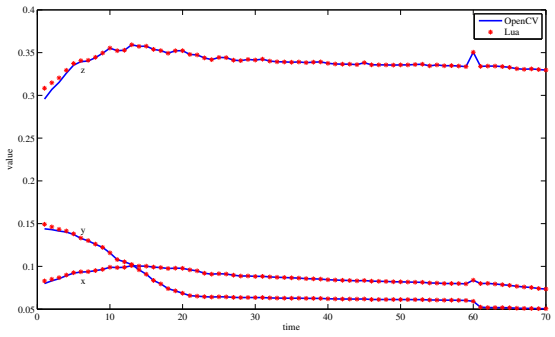

Figure 3: Location comparison of video period 1

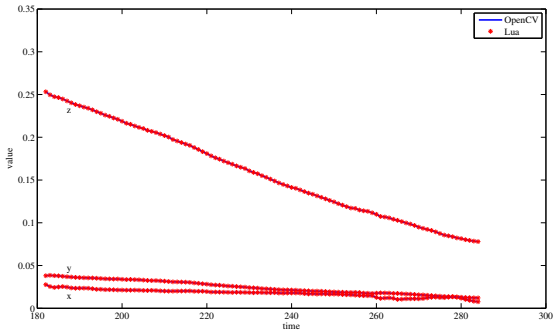

Figure 4: Location comparison of video period 2

As we can see from Figure 3, 4, 5 and 6, the result of our solution and OpenCV are nearly the same. Even there is slightly different when the tag enters the image, the error is totally tolerable.

In the terms of time consuming, it would make no sense to compare C code and Lua code, so we give the chart showing iteration times in our solveQuad function here, as shown in Figure 7. We can see that it takes approximately 6 iterations to get a solution of 0.0000000001 accuracy.

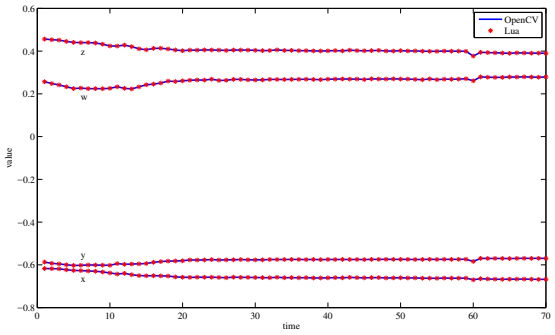

Figure 5: Quaternion comparison of video period 1

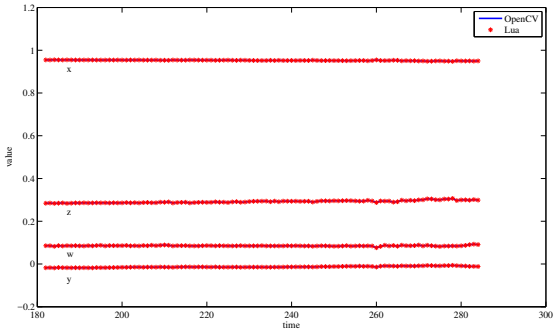

Figure 6: Quaternion comparison of video period 1

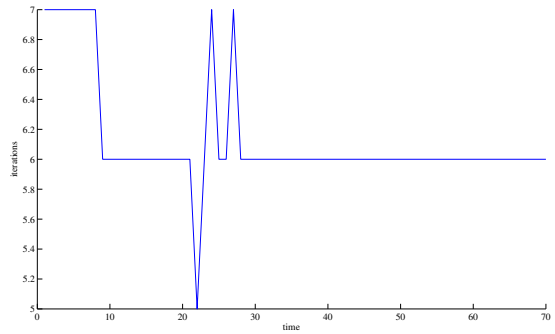

Figure 7: Iterations times in solveQuad function

Aside of results above, we have observed some time that opencv is giving us an unstable result. When the Apriltag is far away enough to the camera, or in a certain pose, the result that opencv is giving becomes very unstable, as shown in Figure 8. We captured pictures of the fault situation, shown in Figure 9, and try to give an explanation. It seems that in certain distance and poses, there may be some errors on the corner location in the picture, and OpenCV is giving another solution of the possible poses. However, in our method, because we are using an initial point calculated by all the camera equations to approach the right solution, we can always get the right solution.

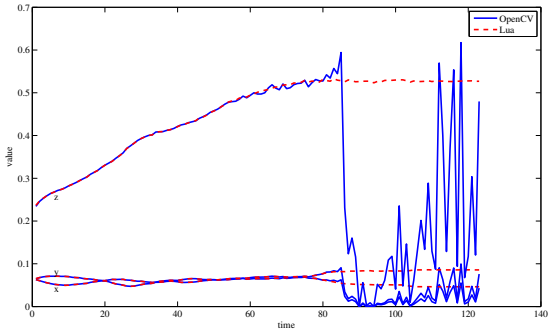

Figure 8: Comparation between Opencv and our solution

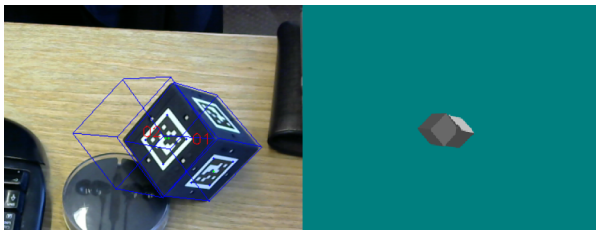

Figure 9: A fault picture

Therefore, we draw the conclusion that our proposed solution fits our requirement, and in some terms outcomes OpenCV's solvePnP with respect t o stability. Further testing with a C implementation is required to compare performance.

References

[1] [apriltag-devel] solvepnp vs. homography_to_pose (accuracy testing). https://april.eecs.umich.edu/pipermail/apriltag-devel/2016-December/000067.html.

[2] Camera calibration and 3d reconstruction. `https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html`. updated on Dec 13, 2017.

[3] Xiao-Shan Gao, Xiao-Rong Hou, Jianliang Tang, and Hang-Fei Cheng. Complete solution classification for the perspective-three-point problem. *IEEE transactions on pattern analysis and machine intelligence*, 25(8):930–943, 2003.

[4] Edwin Olson. Apriltag: A robust and flexible visual fiducial system. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 3400–3407. IEEE, 2011.