Three-dimensional pose estimation of objects using computer vision

Allan Vériter

October 2025

1 Introduction

This paper describes a practical method to estimate the 3 dimensional pose of an object using cameras. The pose refers to the position and orientation of an object. This information is essential in robotics and, more generally, in electronics and electromechanics. The pixel coordinates of the tracked object are first detected by analyzing an image of the scene. These pixel coordinates are then transformed into rays that connect the object to the camera. By finding the intersection of the rays of different cameras, we can estimate the pose of an object. Finally, the most important optimization methods will be discussed. Although this paper describes the mathematical concepts of the method, some code snippets will be included. The complete Python code will be available on the following GitHub page: `https://github.com/allanveriter/pose-estimation`

This paper is the result of a student job carried out at the Robotics Lab of the Vrije Universiteit Brussel (VUB) during August 2024 and August 2025. I would like to thank assistants Ruben Spolmink and Marco Van Cleemput for their valuable help and guidance. I am also deeply grateful to Prof. Dr. Jan Lemeire for providing me with the opportunity to work in the Robotics Lab and for allowing me to develop tools that support bachelor's theses, master's theses, and research activities.

2 Detecting pixels out of a raw image

To easily detect an object with a camera, a fiducial marker has to be put on that object. Two different quadrilateral-shaped markers will be described in this section.

One pattern is an ArUco marker. The other pattern is a set of colored dots.

The algorithm and performed calculations will be described for two cameras that can see the object. Later, techniques to integrate more than two cameras are described. A camera captures frames at a speed that is called frames per second, FPS. The period T between two frames is equal to $\frac{1}{FPS}$. In each period, each camera outputs an image. That image is used to find the pixels of the pattern.

2.1 ArUco marker

An ArUco marker consists of a black square with a unique binary pattern inside.

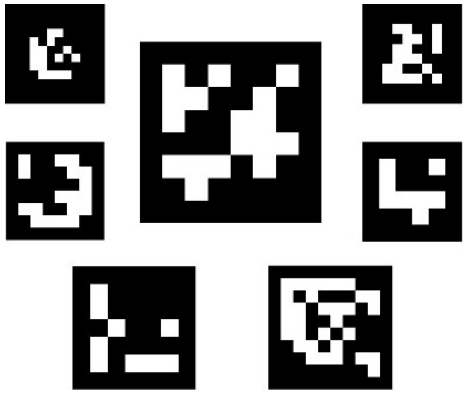

Figure 1: Example of different ArUco markers [1]

An ArUco marker has to be put on the object whose pose needs to be estimated. The marker has to be flat. The marker has to be fully visible by at least 2 cameras. The following OpenCV function returns the pixel coordinates of the corners of the ArUco marker.

```
cv2.aruco.detectMarkers()
```

[1]
For each of the two cameras, there is a set of four pixel coordinates.

$$s_i = (P_1, P_2, P_3, P_4)_i \quad and \quad P_j = (p_x, p_y)$$

for $i = 1, 2$. p_x and p_y are integers between 0 and respectively the width and the height of the camera's resolution. It is important to note that P_1 from camera one and P_1 from camera two point to the same physical corner. The OpenCV function always returns the pixel coordinates in the same order.

2.1.1 Improvement on ArUco method

The drawback of this method is its slowness. Searching for an ArUco marker is slow. The time that this will take depends on the processing speed of the computer used. Though, the process to look for the ArUco marker takes much more computational power than the calculations in the following sections; they are mostly matrix calculations. Besides the bottleneck of 30 fps cameras, which means the algorithm cannot go faster than $\frac{1}{30}$ s per frame, a slow computer will have an even bigger bottleneck when searching for the ArUco marker.

To solve this problem, a region of interest (ROI) will be implemented. The smallest bounding box that contains all four corners of the ArUco marker will be saved. An additional margin of n pixels will be added around each side of the bounding box. The amount of pixels has to be carefully chosen.

An example value will be calculated here for illustration. Take an object moving at 5 kph filmed by a 30 fps camera whose resolution in the width is 1920 pixels. The last element needed for the calculation is the longest distance that the camera can see. This could be the ground, where 3 meters in width can be seen by the camera.

Between two frames, the object will move:

$$\frac{1}{30}\,\mathrm{s} \cdot \frac{5}{3.6}\,\frac{\mathrm{m}}{\mathrm{s}} = 4.6 \cdot 10^{-2}\,\mathrm{m}.$$

The camera will capture this displacement as:

$$\frac{4.6 \cdot 10^{-2}\,\mathrm{m}}{3\,\mathrm{m}} \cdot 1920\,\mathrm{pixels} \approx 30\,\mathrm{pixels}.$$

Therefore, the ROI will be the bounding box with an additional 30 pixels on each side. During the next iteration, the algorithm will try to detect the ArUco marker inside the ROI. If this works, the ROI will be updated with the new bounding box. If this fails, the algorithm will reset the ROI and look at the whole image.

2.2 Colored dots

Another type of marker for detection by computer vision is colored dots. Four colored dots must be placed on the object. The shape does not have to be a dot; any shape can be used. The four dots cannot all have the same color; the computer must be able to distinguish the dots from each other. Two separate colors are enough for the computer to distinguish the four dots. The colors have to be ordered as shown in Figure 2:

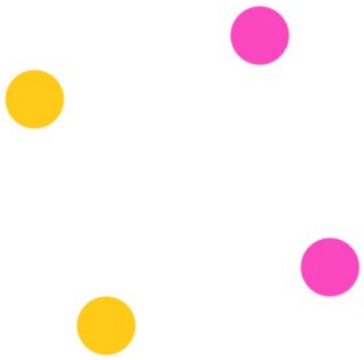

Figure 2: Four colored dots, two separate colors.

The shape formed by the four dots can be any quadrilateral, as long as when looping through the dots clockwise around the center (defined as the average position of all the dots) the first two encountered have the same color, followed by the two dots of the other color.

2.3 The algorithm

The algorithm starts off with an image of what the camera can see. That image is generally in RGB (red–green–blue). This must be converted to the HSV (hue–saturation–value) color model first [6]. Hue is the type of the color, and also the component of the color on which the filtering is going to happen. Saturation and value can vary through different environments, as well as through changes in light intensity. Hue is much more stable. An HSV diagram is shown in Figure 3.

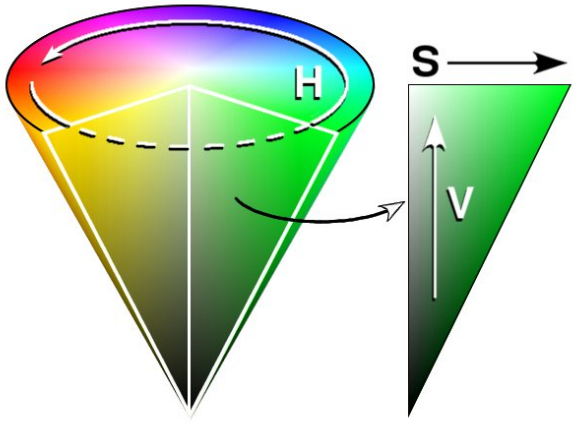

Figure 3: Hue, saturation, value-diagram. From wikipedia. [10]

The hue values of the two colors that will be detected must be known. A small interval in which the hue value lies is enough for the algorithm to work. For each hue color interval, closed contours that are filled with the color are extracted from the image. These contours are ordered by area. The two contours with the largest area are supposed to be the two colored dots that were sought. The middle of these areas is the pixel coordinate that will be associated with that colored dot. P and Y are the pixel coordinates of the pink and yellow dots, respectively. This leaves us with the set of pixel coordinates given by:

$$s = (P_1, P_2, Y_1, Y_2) \quad and \quad P_j \text{ or } Y_j = (p_x, p_y) \quad j = 1, 2$$

There is a problem with this representation. P_1 and P_2 are interchangeable. They could be switched if one dot becomes larger than the other, which can happen when the marked object moves. The colored dots must therefore be ordered clockwise so that their representation in the set becomes unique.

To order the pixel coordinates, the center point C of all four dots is calculated first. This is simply done by taking the average of all pixel coordinates:

$$C = \frac{P_1 + P_2 + Y_1 + Y_2}{4}$$

We first look at the two pink dots. We translate their coordinates so that the center point becomes the origin. This gives two vectors:

$$\vec{v_1} = P_1 - C, \quad \vec{v_2} = P_2 - C$$

Now we can check in which order they lie around the center. This is done by taking the cross product:

$$\vec{v_1} \times \vec{v_2}$$

Since all points lie on the same plane, we only look at the z-component of the result. If the z-component is negative, then P_1 and P_2 are in clockwise order. If the z-component is positive, they are in counter-clockwise order, which means we need to switch P_1 and P_2.

The same procedure is applied to the yellow dots. We calculate:

$$\vec{w_1} = Y_1 - C, \quad \vec{w_2} = Y_2 - C$$

Then we compute the cross product $\vec{w_1} \times \vec{w_2}$. Again, if the z-component is positive, Y_1 and Y_2 must be switched. After this step, both pink coordinates and yellow coordinates are consistently ordered clockwise around the center. We are left with a unique ordered set of four pixel coordinates

$$s_i = (P_1, P_2, P_3, P_4) \qquad for \;\; i = 1, 2$$

The details are analogous to those in the ArUco marker section.

2.3.1 Improvement on colored dots method

The drawback with this method is different from the ArUco method. Speed is not a bottleneck here, since extracting the contours happens fast. An issue with this method is that objects in the scene that have the same color will also be detected. If these objects are larger than the dots, then their centers will be returned instead of the actual positions of the dots. This can be prevented by ensuring that no other objects with the same color are present in the camera scene, although this can be complicated depending on the context or environment.

A better solution consists of taking a reference image of the environment seen by the camera when the object that must be tracked is not present. Then, for each frame where we try to localize the object, we subtract the reference image from the current frame. Everything will appear black except for the areas where new pixels differ from the reference image. This ensures that other objects with the same color as the colored dots will not be detected.

The concept of subtracting two images works as follows: each pixel is represented by three values (R, G, B). For a pixel at time t_0 and at time t_i, the absolute difference

is computed for each channel:

$$D = (|R_i - R_0|,\ |G_i - G_0|,\ |B_i - B_0|)\,.$$

To decide whether a pixel has changed, the differences can be combined into a single value, for example by summing the channel differences:

$$\Delta = |R_i - R_0| + |G_i - G_0| + |B_i - B_0|.$$

If Δ is greater than a chosen threshold, the pixel from the current frame is kept. Otherwise, the pixel is set to black:

$$\text{output}(x, y) = \begin{cases} \text{pixel}_i(x, y), & \text{if } \Delta > \text{threshold}, \\ (0, 0, 0), & \text{otherwise.} \end{cases}$$

The threshold value can be adjusted depending on how sensitive the detection should be. A higher threshold ignores small variations (e.g., lighting changes).

3 Pixel coordinates to a ray in the camera frame

In this section, the goal is to determine the ray in the camera frame that corresponds to a given pixel coordinate (p_x, p_y). It is important to note that a three-dimensional point cannot be uniquely determined from a single two-dimensional pixel coordinate. All three-dimensional points that lie along the same ray from the camera center are projected onto the same pixel on the image plane. Therefore, instead of looking for one three-dimensional point, we need to find the ray on which all these points lie. This concept can be explained using the pinhole camera model and perspective projection [3] [8], which is illustrated in Figure 4. The picture is taken from MIT lectures about the pinhole camera model. [4]

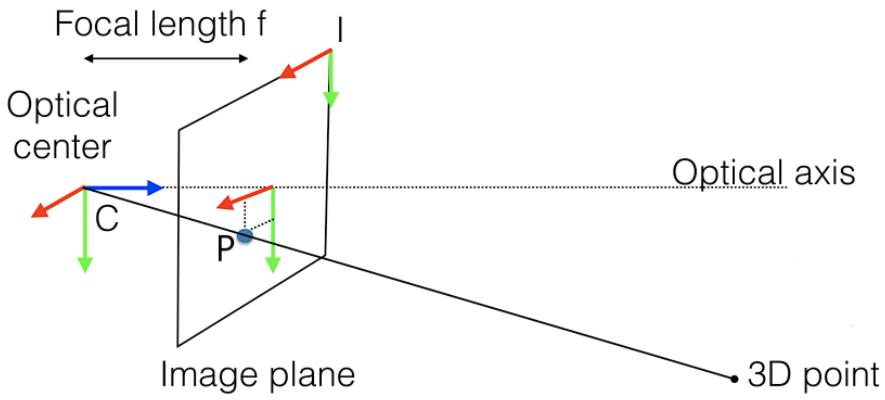

Figure 4: Perspective projection in the pinhole camera model.

A ray in three-dimensional space can be described as a function of a parameter t, with a translation vector $\vec{p_1}$ and a directional vector $(\vec{p_2} - \vec{p_1})$:

$$\vec{r}(t) = \vec{p_1} + (\vec{p_2} - \vec{p_1})t \qquad t \in \mathbb{R}.$$

Since the ray always passes through the origin of the camera frame (the pinhole), $\vec{p_1} = \vec{0}$, and we only need to determine $\vec{p_2}$. We define $\vec{p_2}$ as the intersection of the ray with the image plane. The distance from the pinhole to this plane is the focal length f. To express $\vec{p_2}$ in the camera coordinate frame, the pixel coordinates (p_x, p_y) are first translated so that the optical center (c_x, c_y) becomes the origin, and then scaled with the pixel dimensions ρ_x and ρ_y:

$$\vec{p_2} = \begin{bmatrix} \rho_x(p_x - c_x) \\ \rho_y(p_y - c_y) \\ f \end{bmatrix}$$

In general, ρ_x and ρ_y are approximately the same, but we keep them separate here for completeness. Since $\vec{p_2}$ is a vector, it can be normalized by dividing all its components by f.

$$\frac{\vec{p_2}}{f} = \begin{bmatrix} \rho_x(p_x - c_x)/f \\ \rho_y(p_y - c_y)/f \\ 1 \end{bmatrix} = \begin{bmatrix} \rho_x/f & 0 & -\rho_x c_x/f \\ 0 & \rho_y/f & -\rho_y c_y/f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix}.$$

The directional vector of the ray can be expressed as the multiplication of a vector that contains the pixel coordinates with a matrix. The latter has been extended with a third element equal to one. This is called homogeneous coordinates. Using

homogeneous coordinates makes it possible to describe translations and scalings with simple matrix multiplications. [5]

The matrix on the left describes the scaling between metric units and pixel units in the image plane. If we invert this matrix, we get:

$$\begin{bmatrix} f/\rho_x & 0 & c_x \\ 0 & f/\rho_y & c_y \\ 0 & 0 & 1 \end{bmatrix}.$$

The terms f/ρ_x and f/ρ_y represent the focal lengths expressed in pixels and are denoted as f_x and f_y. Substituting these gives:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}.$$

This is the commonly known camera matrix K [3], which contains the intrinsic parameters of the camera. It converts three-dimensional coordinates in the camera frame to pixel coordinates. To go in the opposite direction, from a pixel to a ray, we use its inverse:

$$K^{-1} = \begin{bmatrix} 1/f_x & 0 & -c_x/f_x \\ 0 & 1/f_y & -c_y/f_y \\ 0 & 0 & 1 \end{bmatrix}.$$

As shown earlier, multiplying the pixel coordinates in homogeneous form by this inverse gives the direction vector of the ray we are looking for:

$$\vec{r}(t) = \vec{v}t = K^{-1} \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} t \qquad t \in \mathbb{R}.$$

This same process is applied to all the pixel coordinates that were previously determined, resulting in a set of four rays for each of the two cameras. A subscript c is added to indicate that the rays are expressed in the camera frame:

$$s_i = (\vec{r_{c1}}, \vec{r_{c2}}, \vec{r_{c3}}, \vec{r_{c4}})_i \quad \text{for } i = 1, 2.$$

4 Camera frame to world frame

The ray $\vec{r}(t)$ derived in the previous section is expressed in the camera frame. It is important to transform the ray to world coordinates because calculations with rays

from different cameras will be done later. All of these cameras are in a different frame. To transition from one frame to the other, a rotation matrix R and a translation vector $\vec{t}$ are needed. These are called the extrinsic parameters of the camera, they describe the orientation and the position of the camera relative to the world origin and can be determined by doing an extrinsic calibration of the camera, but this will not be discussed here. The following relationship exists between the coordinates in the camera frame and in the world frame:

$$\begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + \vec{t}$$

Given a ray $\vec{r_c}$ in the camera frame, its representation in world coordinates is given by:

$$\vec{r}(t) = R^{-1}(\vec{r_c} - \vec{t})$$

The transformation is applied to each ray in the camera frame to obtain a set of four rays in the world frame for the two cameras.

$$s_i = (\vec{r_1}, \vec{r_2}, \vec{r_3}, \vec{r_4})_i$$

for $i = 1, 2$

5 Intersection of two rays

In the ideal case, two rays that point to the same point in the world frame have an intersection point. The problem with computer vision and calculations is the inevitable uncertainties. Quantization uncertainties happen when an image is captured, since pixels are discrete. Each value of the extrinsic or intrinsic parameters also contains an uncertainty. The consequence is that the two rays pointing to the same point will never intersect. A good approximation of the intersection is found by taking the middle of the shortest line segment that connects the two rays. The shortest line segment connecting two non-intersecting rays is always perpendicular on both rays. [2] The two rays whose approximated intersection point is sought will be represented as follows:

$$\vec{r_1}(\lambda_1) = \vec{v_1}\lambda_1 + \vec{c_1}$$
$$\vec{r_2}(\lambda_2) = \vec{v_2}\lambda_2 + \vec{c_2}$$

The direction vector $\vec{n}$ of the line segment connecting these 2 rays can be found by taking the cross-product of the direction vector of the 2 rays:

$$\vec{v_1} \times \vec{v_2} = \vec{n}$$

Any segment connecting the 2 rays can be found by subtracting the first ray from the second. If that segment is a scalar multiple of $\vec{n}$, the shortest line segment is found. The following vector equation in $\lambda_1, \lambda_2, \lambda_3$ must be solved:

$$\begin{aligned}
&\lambda_3 \vec{n} = \vec{r_2} - \vec{r_1}, \\
&\lambda_3 \vec{n} = (\vec{v_2}\lambda_2 + \vec{c_2}) - (\vec{v_1}\lambda_1 + \vec{c_1}),
\end{aligned}$$

$$\begin{bmatrix} v_{1x} & -v_{2x} & -n_x \\ v_{1y} & -v_{2y} & -n_y \\ v_{1z} & -v_{2z} & -n_z \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{bmatrix} = \begin{bmatrix} c_{2x} - c_{1x} \\ c_{2y} - c_{1y} \\ c_{2z} - c_{1z} \end{bmatrix}$$

This system of equations in $\lambda_1, \lambda_2, \lambda_3$ can be solved using various numerical techniques, such as `numpy.linalg.solve()` [9]. The approximation of the intersection R between the two rays is now given by:

$$R = \frac{(\vec{v_2}\lambda_2 + \vec{c_2}) + (\vec{v_1}\lambda_1 + \vec{c_1})}{2}$$

Performing these calculations for each pair of rays gives us a set of four three-dimensional points:

$$s = (\vec{R_1}, \vec{R_2}, \vec{R_3}, \vec{R_4},)$$

6 Estimating the pose

We assume that the rigid object has known coordinates in its own object frame, with its center defined as the origin. Let us denote four key reference points on the object by

$$\vec{Q_1},\ \vec{Q_2},\ \vec{Q_3},\ \vec{Q_4}$$

where each is expressed relative to the object center at $(0, 0, 0)$. These points are known from the object model in its local coordinate system. For instance, an ArUco marker whose side is 10 cm long will have the following points as reference points:

$$(-0.05, 0, -0.05),\ (0.05, 0, -0.05),\ (0.05, 0, 0.05),\ (-0.05, 0, 0.05)$$

In parallel, from image measurements and calculations we have obtained a corresponding set of four three-dimensional points

$$\vec{R}_1,\ \vec{R}_2,\ \vec{R}_3,\ \vec{R}_4$$

expressed in the world frame. The goal is now to find the rigid-body transformation between the object and the origin; a rotation matrix R and a translation vector $\vec{t}$ that best align the known object points $\{\vec{Q}_i\}$ with the observed points $\{\vec{R}_i\}$. This can be achieved by minimizing the following expression, also known as a least-squares minimization.

$$\min_{R,\vec{t}} \sum_{i=1}^{4} \left\| R\,\vec{Q}_i + \vec{t} - \vec{R}_i \right\|^2.$$

We seek the pair $(R, \vec{t})$ that minimizes the sum of squared distances between the transformed model points and the measured points in the camera frame. This problem can be solved by applying the Kabsch algorithm. [7]

After applying the algorithm, we obtain the transformation

$$R,\ \vec{t}$$

that aligns the object local model to the observations and calculations made by the cameras. While R is a 3×3 orthonormal matrix, in many practical applications it is convenient to represent the orientation in other forms: for example, one may convert R into Euler angles (roll, pitch, yaw) or into a quaternion representation $\vec{q} = (q_w, q_x, q_y, q_z)$.

7 Multiple cameras (>2)

Two cameras are sufficient to localize an object. However, having more cameras can enhance the reliability or precision of the system. There are two ways of integrating more than two cameras into the system.

The part of the algorithm that must be modified is the step where the data from two different cameras are used to compute the next step. This happens during the intersection of the rays. Instead of taking the intersection of two rays, we have now three different rays. To solve this, the intersection of two rays will be calculated for each different pair. For three rays, there are three possible pairs, thus three different intersections. In general, for n cameras, and thus n rays, there are $\frac{n(n-1)}{2}$ possible intersections. Instead of using a single intersection, we can compute the average of

all possible intersections to obtain a more reliable estimate of the intersection point. Although the exact sources of uncertainty in the process are not fully characterized, it can be assumed that individual uncertainties are approximately independent and unbiased. In this case, positive and negative deviations tend to cancel out when averaged, similar to how the mean of a zero-mean distribution approaches zero as more samples are considered. This approach enhances the precision of the localization.
For the other approach, a sorted list containing all the cameras will be introduced.

$$sorted\ list = [cam_1,\ cam_2,\ ..., cam_n]$$

During each iteration of the tracking process, the algorithm iterates over the list. It has to find 2 cameras that can see the object in order to track the object. When 2 cameras that fulfill this are found, the search is stopped. The two cameras that were found are pushed to the front of the list. This ensures that these cameras will be checked on first during the next iteration of the algorithm. This approach results in a faster system, which may be slightly less precise, but overall more reliable than the previous method. It is faster because searching for the object, certainly with the ArUco method, takes time. It is more reliable because multiple cameras can be installed in the environment where the object must be tracked. Each camera cannot capture the whole scene, thus multiple cameras are needed.

8 Computing uncertainties

There is an uncertainty in the computation of the pose. The uncertainty can come from several sources. The main sources are listed below:

- The marker might be applied imprecisely, leading to an uncertainty in its dimensions and position.
- Quantization uncertainty when the marker is represented by pixel coordinates, since they are discrete and not continuous.
- Uncertainty in the intrinsic parameters of the camera, represented by the camera matrix K.
- Uncertainty in the extrinsic parameters of the camera, namely the rotation matrix R and the translation vector $\vec{t}$.

These uncertainties result in an overall uncertainty in the intersection of the rays. This then propagates to the computation of the translation and pose with the Kabsch

algorithm. One way to estimate the pose uncertainty is by considering the variance of the residuals from the least-squares fit:

$$\sigma^2 = \frac{1}{4}\sum_{i=1}^{4} \left\| R\,\vec{Q}_i + \vec{t} - \vec{R}_i \right\|^2$$

The square root of this variance, σ, gives an estimate of the standard deviation of the pose error.

Another way to think about it is to consider the geometry of the ray intersections. If all rays intersected perfectly, the shortest line segments connecting them would have zero length. Therefore, the average length of the shortest lines connecting each pair of rays provides another reasonable measure of uncertainty:

$$\frac{1}{4}\sum_{i=1}^{4} \left\| (\vec{v_{2,i}}\lambda_{2,i} + \vec{c_{2,i}}) - (\vec{v_{1,i}}\lambda_{1,i} + \vec{c_{1,i}}) \right\|$$

9 Python, libraries and code

The method described in this paper has been fully programmed and tested to ensure its reliability in practical situations. The use of `NumPy`, a Python library for fast matrix and vector calculations, has been a key part of implementing the system. OpenCV is another Python library that has been important, mainly for the computer vision part. Image compression has also been important to maintain a high resolution while keeping enough frames per second.
The complete code is available on the following GitHub page: `https://github.com/allanveriter/pose-estimation`

10 Conclusion

This paper described a complete and practical method to estimate the three-dimensional pose of an object using multiple cameras. Starting from the detection of a known pattern in the images, we converted pixel coordinates into rays, transformed them into the world frame, and found the approximate intersections between corresponding rays. From these intersections, the pose of the object was calculated using the Kabsch algorithm.

Both ArUco markers and colored dot markers were discussed, each with their own advantages and drawbacks. While ArUco markers provide higher robustness, they

are slower to detect. Colored dots, on the other hand, allow for faster processing but require a controlled environment.

Finally, improvements using multiple cameras and ways to estimate uncertainties were presented. The approach shows that reliable three-dimensional pose estimation can be achieved using standard cameras and open-source tools. With proper calibration and optimization, this system can be applied to various robotics or tracking tasks where precise object localization is needed.

References

[1] Detection of aruco markers. OpenCV Documentation. Accessed: 2025-10-31.

[2] The distance between two skew lines. Technical report, University of California, Riverside, 2013. Accessible at https://math.ucr.edu/ res/math133-2018/skew-lines.pdf.

[3] Baeldung. Focal length and intrinsic camera parameters. Baeldung.com. Last updated March 18, 2024.

[4] Luca Carlone. Visual navigation for autonomous vehicles (vnav): Lecture 11: Pinhole camera model (image formation notes). Lecture notes, MIT. Accessible at https://vnav.mit.edu/material/11-ImageFormation-notes.pdf, Fall 2021.

[5] Tom Dalling. Explaining homogeneous coordinates & projective geometry. Tom Dalling Blog. Blog post.

[6] Dijin Dominic. A beginner's guide to understand the hsv color model. Medium Article, 2024. Published Oct. 6, 2024.

[7] Hunter Heidenreich. Kabsch algorithm: Numpy, pytorch, tensorflow, and jax. Hunter Heidenreich Blog. Blog post.

[8] Jan Lemeire. Robot vision. November 2021.

[9] NumPy Developers. numpy.linalg.solve — numpy v1.26 manual. NumPy Documentation, 2025. Accessed: 2025-11-03.

[10] Wikipedia. Hsv (kleurruimte). Wikipedia, n.d. Accessed: 2025-11-05.