Simple and Accurate Pose Estimation Using Unique Markers

Siddharth Trehan

Abstract

I describe a system for visual pose estimation using unique color-coded markers, which I demonstrate to have high accuracy and ability to be run in real-time. First, I make modifications to the commonly-used Aruco tag to make it more suitable for more precise pose estimation while retaining the ability to distinguish between multiple unique tags in the same scene. Next, I implement a new algorithm for PnP (2D to 3D points correspondence) to allow for much more stable pose estimation of planar markers. Finally, I quantify the errors associated with pose estimation using these techniques and show how it can be integrated in an existing pose estimation setup using SLAM.

Keywords: Computer Vision, Pose Estimation, PnP

1. Introduction

Pose estimation is the problem of determining the pose of an object in a scene. Pose often includes 6 degrees of freedom: 3 for translation, and 3 for rotation, though in some cases 6 additional degrees of freedom for scaling and shear are also included. Here, we are concerned with determining the pose of rigid objects, so the problem is to estimate the translation and rotation of an object from a model of that object in a scene. Furthermore, pose estimation can be accomplished using a variety of techniques such as range sensors and structured light projectors and scanners. It is often the case, however, that we wish to accomplish pose estimation using an optical camera, because it is powerful enough, small enough, and cost-effective enough to accomplish a number of other tasks in addition to just pose estimation. And finally, in some settings we are to determine the pose of an unknown object, whereas in other settings we have the freedom to design the object to recognize. The latter is applicable, for example, in factory settings, where we can design

an easily detectable "marker" which will allow us to estimate the pose of whichever object the marker is attached to. This marker is often a planar tag so that it can be rendered more easily. In this work, we are concerned with the rigid pose estimation of a specifically designed planar marker using an optical camera.

Aruco tags are often the go-to marker for pose estimation. Each Aruco tag has a unique ID and easily detectable corners, and so an Aruco tag detection algorithm can report the detection of several Aruco tags in a scene by their ID and identify where their corners are. This is helpful when there are multiple objects to be detected in a scene, or where a single object is associated with multiple Aruco tags at different orientations to allow for pose estimation from a much larger field of view. Furthermore, because the Aruco tag is so simple and unique, it has high probability of detection. Examples of Aruco tags with several different IDs are shown in Figure 1. Detection of Aruco tags has support in several different programming languages including C++ and Python, and is available in the popular OpenCV library.

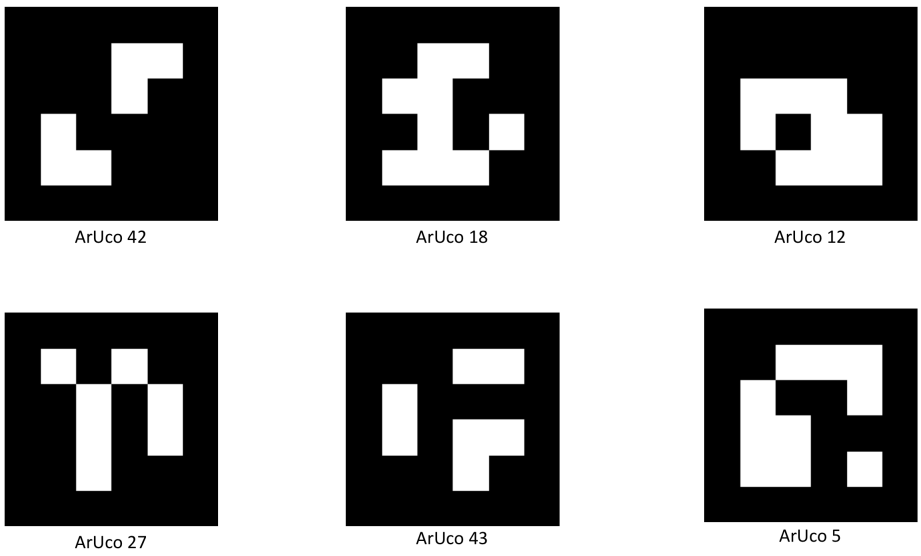

Figure 1: Examples of Aruco tags of several different IDs

However, an important drawback is that pose estimation from Aruco tags is not always as accurate as required. Due to noise from a combination of electrical, optical, and structural factors, corner detection can only be

accurate down to within ± 1 pixel. This produces noise in the pose estimated from the locations of these detected corners. We can use the central limit theorem to estimate that if the variance of the estimated pose due to the noise in one detected corner is Σ, then the variance of the estimated pose due to the noise in N detected corners is $\frac{1}{N}\Sigma$. Therefore, we can increase the accuracy of pose estimation by increasing the number of detectable corners.

One way to do this is by using multiple redundant Aruco tags. Indeed, one popular method is the use of a checkerboard of Aruco tags, called a Charuco board (as shown in Figure 2). However, the amount of space this takes up is unreasonable for many applications. Therefore, one problem I seek to address in this work is how to design markers with high detection probability and large number of corners relative to the space that the marker occupies.

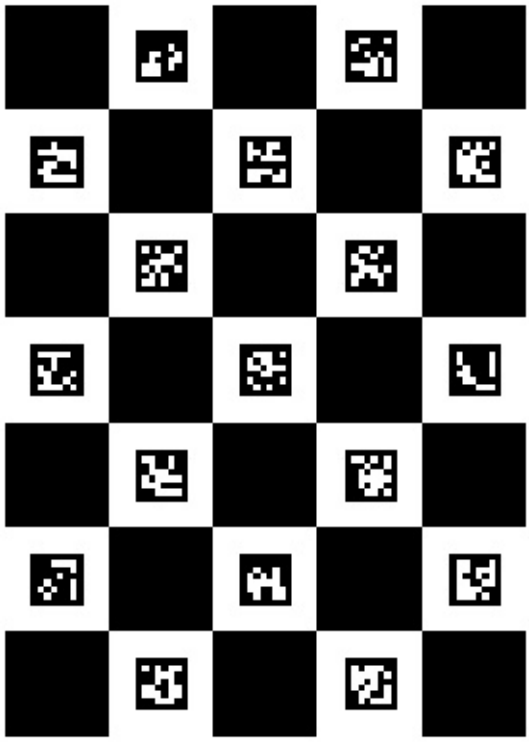

Figure 2: An example of a Charuco board

Another problem emerges because having all detectable corners in the same plane of the 3D model presents challenges to most existing PnP pose

estimation algorithms, particularly when the marker is oriented so that the normal vector to its plane is oriented close to parallel to the normal vector of the image plane of the camera. This often manifests as a bistability, where a small amount of noise in the detected points can cause the PnP algorithm to converge to one of two very different poses. Coplanar corners, however, are a very common case for markers because the marker can then be printed with an ordinary desktop printer. This is another problem I address here: the design of a PnP algorithm that does not have any bistable conditions and is accurate in the pose estimation of a marker of coplanar points.

The last problem I address stems of the characterization of noise in this pose estimation approach. Because the frame rate of the optical camera is often not as high as we would like it to be, there is an interpolation step to estimate the most likely poses between the sampled timesteps. Furthermore, fluctuations in pose between timesteps can be smoothened out by assuming a probability model for the motion of the object in close timesteps. Finally, by quantifying the error, we can incorporate the camera and its pose estimation algorithm as another sensor in SLAM.

2. Marker Design

I expand on the Aruco tag to create a marker that is uniquely identifiable by its ID and is simple and unique enough to have high detection probability, but also optimizes the ratio of number of corners to space usage. I propose the "pixelcode", which exploits color to present different attributes in different color channels, as shown in Figure 3. To demonstrate why this works, we must first understand the process of Aruco tag detection, which is summarized here:

1. The borders of the Aruco tag are detected by finding edges and corners which have high contrast with their background and are approximately a quadrilateral (it is for this reason Aruco tags must have some white padding surrounding them).
2. The four corners of the Aruco tag are identified and used to identify a prospective transform matrix M.
3. The matrix M is then used to find the approximate corner points of the inner grid carrying ID information.
4. Each cell in the inner grid is determined to be either majority white or majority black. If any cell is ambiguous, the algorithm will fail to detect an Aruco tag.

5. Finally, the ID of the tag is calculated from the pattern of black and white grid cells, and the ID and the pixel locations of the detected corners are returned.

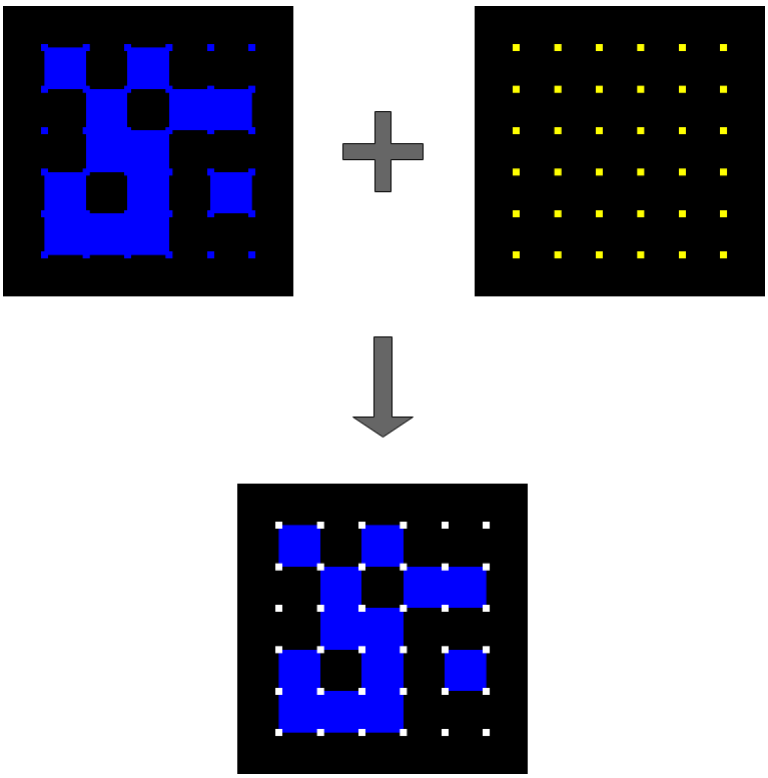

Figure 3: An example of a pixelcode and its view in the two different color channels of interest

In the pixelcode, the intuition is to separate an Aruco tag and a grid of dots into separate color channels, taking caution to minimize interference between the two channels. The Aruco tag can be used to determine ID and disambiguate orientation, and the grid of dots provides many corners to improve accuracy of pose estimation. In the color channel where the Aruco tag is primarily captured, the dots will be slightly visible, but because the dots are so small they will not interfere will the determination of the inner grid colors in the calculation of Aruco tag ID. In the other color channel where the dots are primarily captured, the partially visible corners from the inner grid of the Aruco tag will not interfere with the detection of corners

because the corners of the inner grid of the Aruco tag all coincide with the locations of the dots.

The main constraint to consider when choosing the colors of the pixel code is the signal to noise ratio in the color filter for finding dots. The probability of detection of any dot will increase with the brightness of the dot under this color filter. Because the color filters in a camera are not orthogonal and there exists some overlap for which one color will be faintly detected in another color filter, it is optimal to make the dot bright in every color filter. This will not interfere with Aruco tag detection because the dots can be made arbitrarily small.

A further consideration is how to do dot detection. The steps in dot detection can be roughly broken down into the following:

1. Apply an image smoothing filter to suppress false corners and apply a corner detection algorithm to detect dots.
2. Remove falsely detected dots, and fail to detect the pixelcode if not all true dot are found.
3. Determine the row-wise and column-wise ordering of the dots and return them in this order so that correspondences can be made between dot projections and the 3D dot model.

One common algorithm for corner detection in step 1 is Shi-Tomasi, which determines if both eigenvalues of the image gradient intensity matrix surpass a given threshold. This works well in detecting all corners and occasionally returns some false positives– however, for our application, it rarely misses corners when corners truly do exist. Furthermore, each dot is of a non-zero size and actually has four corners, so non-maximal suppression is required to ensure dots that are close enough together register as only one dot. The image should also be pre-processed with a median filter so that all impulses smaller than a certain size are removed while preserving the edges on the dots.

Steps 2 and 3 can be combined by taking information from the corners registered by the Aruco tag detection algorithm to inform the suppression of false corners and the ordering of detected true corners. To do this, first an approximate projection matrix M is calculated from the four corners of the Aruco tag. Then, M is used to compute where the dots would be if M were the correct projection matrix. Then, the nearest detected corner to each projected corner is found and returned. In this way, dots that are too far

from where the dots should ideally be are rejected, and the dots are returned in the correct order. Furthermore, if two nearest neighbors happen to coincide, which would happen if one dot was not detected altogether, then the algorithm rejects the pixelcode instead of returning faulty dot coordinates. I've implemented nearest neighbor matching with the efficient KDTree data structure and found it works well in real-time dot registration.

In this way, a tag with a uniquely distinguishable ID, high probability of detection, high accuracy, and low space usage can be designed, and an algorithm for its detection can be implemented.

3. PnP Algorithm

Most PnP algorithms (including all the ones provided in the OpenCV library) have trouble estimating the pose of coplanar points, even though for convenience reasons most markers consist of coplanar points. I observed that when the marker was positioned head-on to the camera (so that the normal vector to the marker was close to alignment with the normal vector of the image plane) and both the camera and the marker were kept stationary, the estimated poses neatly fell into two clusters. The translational distance between these two clusters could be up to 7.5 centimeters apart (using a camera of focal length 2.1 millimeters). This bistability was preserved even after adjusting for camera distortion as well as slight bending of the tag. An example of this bistability occurring is shown in Figure 4.

Figure 4: An example illustrating the bistability of pose estimation when observing a stationary tag– a small deviation due to noise causes the estimated pose to flicker drastically

Bistability is produced by of a combination of two factors: the coplanarity

of points produces a singular matrix, and non-linearity creates the possibility of more than one local minimum. Therefore, the ideal PnP algorithm should not invert a singular matrix and should be linear. Ideally, it should also be based on a probabilistic model, where the projections of corners onto the image plane can be off by a small amount in any direction– however, the algorithm I propose does not have this property because it is more important that the algorithm be linear and not invert a singular matrix.

Consider the perspective projection model as shown in Figure 5. In this model, each point from the 3D model is projected onto the image plane via a line from that point to the focal point of the camera (which we can from now on take as the origin of our coordinate system). Therefore, a point in the real world $\vec{R}_i$, its projection onto the image plane $\vec{r}_i$ (which is what we observe in our camera), and the focal point at the origin are all collinear.

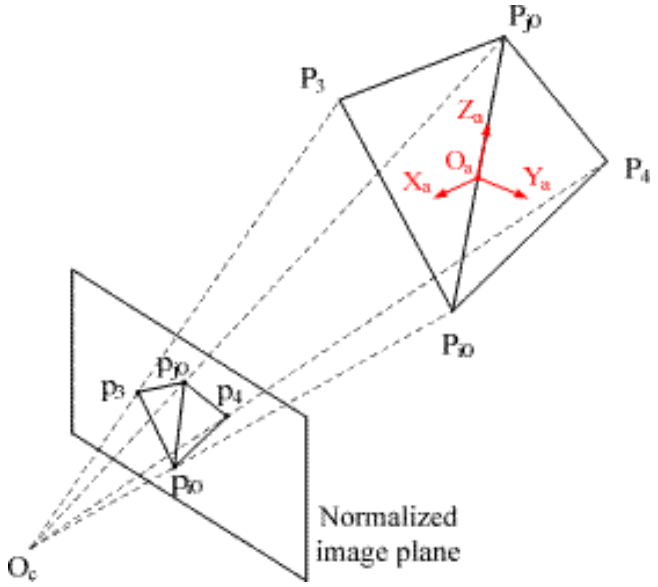

Figure 5: The perspective projection model

One way to state that three points $\vec{a}$, $\vec{b}$, and $\vec{c}$ are collinear is the statement that the cross product of the displacement vectors between them is zero:

$$(\vec{a} - \vec{b}) \times (\vec{a} - \vec{c}) = 0 \tag{1}$$

In our case, one of the points is the focal point, which we take as our origin, so:

$$\vec{r_i} \times \vec{R_i} = 0 \tag{2}$$

The vector $\vec{r_i}$ should have as its z-coordinate f, the focal length of the camera. Therefore, $\vec{r_i} = \begin{bmatrix} x_i \\ y_i \\ f \end{bmatrix}$, where x_i and y_i are the pixel coordinates of the point we observe on the image, taking the center of the image as the origin. We can reformulate this in terms of the matrix form of the cross product, which says:

$$\vec{a} \times \vec{b} = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix} \vec{b} \tag{3}$$

Therefore, if we define:

$$X_i = \begin{bmatrix} 0 & -f & y_i \\ f & 0 & -x_i \\ -y_i & x_i & 0 \end{bmatrix} \tag{4}$$

Then we can reformulate Equation 2 as a linear matrix equation in terms of a known X_i and an unknown $\vec{R_i}$. Furthermore, given that all the points in the 3D model are coplanar, we can rewrite every $\vec{R_i}$ as a linear combination of two basis vectors $\vec{u}$ and $\vec{v}$ and a translation vector $\vec{t}$: $\vec{R_i} = a_i\vec{u} + b_i\vec{v} + \vec{t}$. We can combine this insight with Equation 2 and Equation 4 to get the following linear equation:

$$a_i X_i \vec{u} + b_i X_i \vec{v} + X_i \vec{t} = 0 \tag{5}$$

Where $\vec{u}$, $\vec{v}$, and $\vec{t}$ are the parameters to estimate and the other terms can be calculated from observed information. There are n such points in the model, and for each one we have one such equation, so written in block matrix form, the overall equation is:

$$\begin{bmatrix} a_1X_1 & b_1X_1 & X_1 \\ a_2X_2 & b_2X_2 & X_2 \\ \vdots & \vdots & \vdots \\ a_nX_n & b_nX_n & X_n \end{bmatrix} \begin{bmatrix} \vec{u} \\ \vec{v} \\ \vec{t} \end{bmatrix} = 0 \tag{6}$$

This model can be solved using a least-squares method. There are a few things to note about this model:

1. There are a total of nine free parameters to estimate. This translates to three translational distances, three rotational angles, two scaling factors, and one shear factor.
2. Because we are dealing only with rigid body pose estimation, we only want to estimate six free parameters: three translational distances and three rotational angles. However, constraints on unitary scaling and no shear are generally non-linear.
3. The matrix on the left-hand-side of Equation 6 cannot have a rank of 6, and therefore all solutions lie along a line. This has a intuitive interpretation: we are unable to distinguish an object x inches tall y inches away from an object $2x$ inches tall $2y$ inches away. If we could constrain $\vec{u}$ and $\vec{v}$ to be unit vectors, we could solve this issue, but that would introduce a non-linear constraint.

To be able to solve this equation down to a single point, therefore, a new linear constraint should be introduced. One simple solution is to constrain the z-component of $\vec{t}$ to be a constant amount, and then after solving the linear equation figure out which scaling factor produces unit vectors for $\vec{u}$ and $\vec{v}$. This means we modify Equation 6 in the following way:

$$\begin{bmatrix} a_1X_1 & b_1X_1 & X_1 \\ a_2X_2 & b_2X_2 & X_2 \\ \vdots & \vdots & \vdots \\ a_nX_n & b_nX_n & X_n \\ 0 & 0 & 0 \quad 0 \quad 1 \end{bmatrix} \begin{bmatrix} \vec{u} \\ \vec{v} \\ \vec{t} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \tag{7}$$

$\vec{u}$ and $\vec{v}$ will not have exactly the same magnitude, and they will also not be exactly orthogonal. One simple procedure we can after running least squares on the system in Equation 7 is to arbitrarily choose one of them as being "exactly correct" and orthonormalize the other with respect to the first using Gram-Schmitt. This eliminates both scaling and shear and allows us to solve a linear equation exactly to one point, but it does require us to arbitrarily accept one of $\vec{u}$ or $\vec{v}$ as having no noise. In practice, however, this works fine since $\vec{u}$ and $\vec{v}$ tend to be so nearly orthogonal and so similar in magnitude. Further refinements to this step of the model can be made after the we characterize the error in Section 4.

4. Error Quantification

We can make further refinements in our algorithms, do pose interpolations across time, and integrate our pose estimation algorithms into SLAM by characterizing the error of the PnP algorithm from Section 3. This will also help us understand the robustness of our PnP algorithm.

To do this, let's suppose that when we observe $\vec{r}_i$ we have some error in all three dimensions: error in pixel coordinates due to different sources of noise as well as some error in the calculated focal length of the camera due to some possible inaccuracy in calibration. What will be the resulting errors in our estimated $\vec{u}$, $\vec{v}$, and $\vec{t}$ (for now ignoring the subsequent orthonormalization step)?

Let $\vec{y}$ be a vector of length $2n+1$ containing $2n$ pixel coordinates of all the points on the image plane as well as 1 component for the focal length of our camera. Let us assume that our 3D model (which contains a_i and b_i for every point) is exact and has no error, which is often the case given our rendering of the marker is accurate enough. Furthermore, let us assume that the error in $\vec{y}$ is Gaussian so that we can better leverage linearity. Then, the matrix A on the left-hand-side of Equation 7 can be completely calculated from $\vec{y}$ and knowledge of the 3D model (which, because there is no error in it, is treated as a constant). Every entry of A is actually linear in $\vec{y}$, and so the matrix A can be expressed as a function of y:

$$A = F\vec{y} + A_0 \tag{8}$$

Since A is an $(3n+1)\times 6$ matrix, and $\vec{y}$ is a $(2n+1)\times 1$ vector, F must be a $(3n+1)\times 6\times(2n+1)$ tensor, and the multiplication here is a generalization of ordinary matrix multiplication: the elementwise product followed by a sum along the common dimension. Therefore, a tensor of dimension d_1 and a tensor of dimension d_2 will multiply to produce a tensor of dimension d_1+d_2-2. The error model implies that errors in $\vec{y}$ will produce errors in A, which will then produce errors in $\vec{x}$, so we should first determine how errors in $\vec{y}$ relate to errors in A. We can then express the covariance of A (denoted from here on as $\Sigma(A)$) with respect to $\Sigma(\vec{y})$:

$$\Sigma(A) = F\Sigma(\vec{y})F^{\mathrm{T}} \tag{9}$$

$\Sigma(A)$ is a $(3n+1)\times 6\times 6\times(3n+1)$ tensor, and since the transpose operation on a tensor inverts the dimensions, F^{T} is a $(2n+1)\times 6\times(3n+1)$ tensor.

As in the rest of this work, we are still dealing with the aforementioned generalization of ordinary matrix multiplication.

$\Sigma(\vec{y})$ is typically a diagonal matrix since the errors in different pixel coordinates, and the focal length, are usually uncorrelated with each other. Furthermore, the error in pixel coordinates can be taken to be ± 1 pixel, while the reprojection error from the camera calibration process can be used to inform the error in focal length.

Meanwhile, F can be calculated from the structure of the cross product matrix X_i and the structure of the block matrix A. Using δ_i to denote one-hot vectors at position i, X_i can be expressed as $X_i = G_i\vec{y}$, where G_i is the following tensor:

$$G_i = \begin{bmatrix} 0 & -\delta_3 & \delta_2 \\ \delta_3 & 0 & -\delta_1 \\ -\delta_2 & \delta_1 & 0 \end{bmatrix} \begin{bmatrix} \delta_{2i-1} \\ \delta_{2i} \\ \delta_n \end{bmatrix} \tag{10}$$

The first tensor on the right-hand-side indicates constructing the cross product matrix, and the second tensor indicates selecting the right components of $\vec{r}_i$ from the $\vec{y}$ vector. Furthermore, A can be expressed as the following function of the X_i matrices:

$$A = \begin{bmatrix} a_1\delta_1 & b_1\delta_1 & \delta_1 \\ a_2\delta_2 & b_2\delta_2 & \delta_2 \\ \vdots & \vdots & \vdots \\ a_n\delta_n & b_n\delta_n & \delta_n \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_n \end{bmatrix} + A_0 \tag{11}$$

Where $A_0 = \delta_{3n+1\,9}$, establishing the rule that the z-component of the $\vec{t}$ vector should be fixed (this is a detail that will not be needed for the final result). This can then be combined with Equation 10 to produce an equation of the form $A = F\vec{y} + A_0$ where:

$$F = \begin{bmatrix} a_1G_1 & b_1G_1 & G_1 \\ a_2G_2 & b_2G_2 & G_2 \\ \vdots & \vdots & \vdots \\ a_nG_n & b_nG_n & G_n \\ 0 & 0 & 0 \end{bmatrix} \tag{12}$$

Now we can compute the relationship between $\Sigma(\vec{x})$ and $\Sigma(\vec{y})$. To do this,

we first compute $\frac{\partial \vec{x}}{\partial A}$, and then multiply by $\frac{\partial A}{\partial \vec{y}} = F$. Then, we can compute $\Sigma(\vec{x}) = \left(\frac{\partial \vec{x}}{\partial \vec{y}}\right) \Sigma(\vec{y}) \left(\frac{\partial \vec{x}}{\partial \vec{y}}\right)^{\mathrm{T}}$. First:

$$\begin{aligned} Ax &= b \\ \frac{\partial}{\partial A}(Ax) &= \frac{\partial b}{\partial A} \\ x + A\frac{\partial \vec{x}}{\partial A} &= 0 \\ A\frac{\partial \vec{x}}{\partial A} &= -x \end{aligned} \tag{13}$$

We can substitute our least-squares solution from Equation 7 for x and solve this linear equation using least-squares to find $\frac{\partial \vec{x}}{\partial A}$. Next:

$$\begin{aligned} \Sigma(\vec{x}) &= \left(\frac{\partial \vec{x}}{\partial \vec{y}}\right) \Sigma(\vec{y}) \left(\frac{\partial \vec{x}}{\partial \vec{y}}\right)^{\mathrm{T}} \\ \Sigma(\vec{x}) &= \left(\frac{\partial \vec{x}}{\partial A}F\right) \Sigma(\vec{y}) \left(F\frac{\partial \vec{x}}{\partial A}\right)^{\mathrm{T}} \end{aligned} \tag{14}$$

Which we can now compute entirely because we know F, $\frac{\partial \vec{x}}{\partial A}$, and $\Sigma(\vec{y})$.

We now know that because our model is linear in $\vec{y}$, the error in $\vec{x}$ is a Gaussian with zero mean and covariance computable using Equation 14.

5. PnP Refinement

In Section 3, we computed the least-squares solution at a fixed scale and then orthonormalized one vector with respect to the other using Gram-Schmitt, arbitrarily taking one to be completely correct. We can now use the results from Section 4 to refine our pose estimation.