TANGRAM VISION

CALIBRATION DESK REFERENCE

TUTORIALS, REFERENCES, & RESOURCES

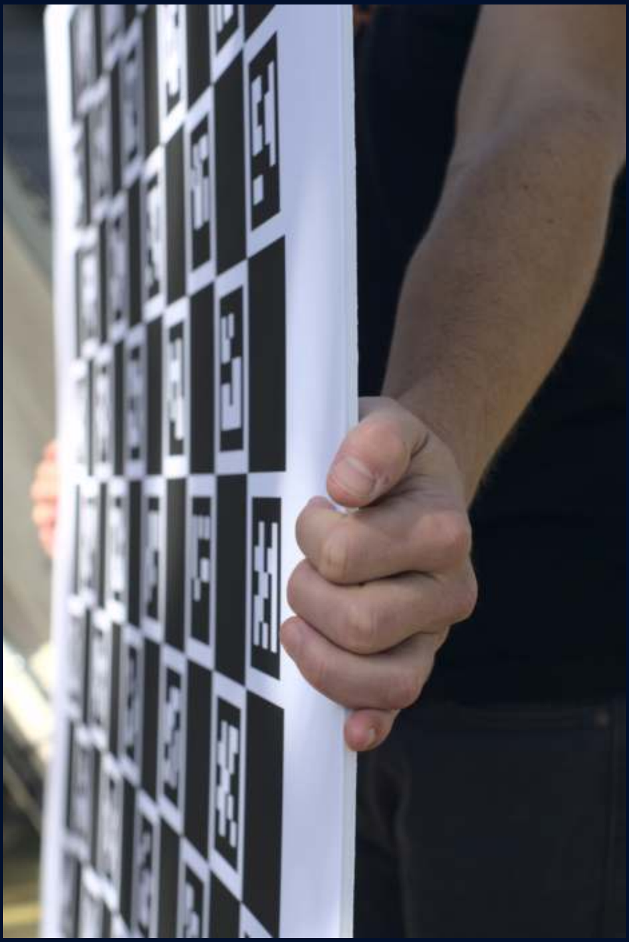


TABLE OF CONTENTS

3 / About Tangram Vision

4 / Section I: Calibration Theory & Model Formation

46 / Section II: Calibration Implementation

66 / Section III: Understanding Calibration Results

74 / Appendix: Resource Library

ABOUT TANGRAM VISION

Tangram Vision is **Perception DevOps**. Our platform helps robotics and autonomy engineers manage critical perception tasks like sensor fusion, integration, calibration, and maintenance as their fleets scale.

Just like you, we're perception engineers. Our team has built products that use sensors, and we've also built perception sensors, too. Over our years of working in and around perception, we came to realize that there was something missing in the world of perception - a ready-to-deploy software stack to manage the foundations of rock solid perception. So that's exactly what we've built!

The Tangram Vision Platform includes tools for multimodal calibration, sensor fusion, perception system stability, sensor integration, and sensor data management.

It supports cameras of all types (including HDR and thermal), depth sensors, LiDAR, and IMUs. Additional modalities including radar and ultrasonic are currently under development.

We encourage you to sign up for an account at hub.tangramvision.com and test the Platform with your own perception pipeline.

And, if you are as deeply fascinated with perception as we are, we invite you to visit our careers page at www.tangramvision.com/careers to explore opportunities to join our team.

SECTION I: CALIBRATION THEORY & MODEL FORMATION

5 / Camera Modeling: Focal Length & Colinearity

12 / Camera Modeling: Exploring Distortion and Distortion Models

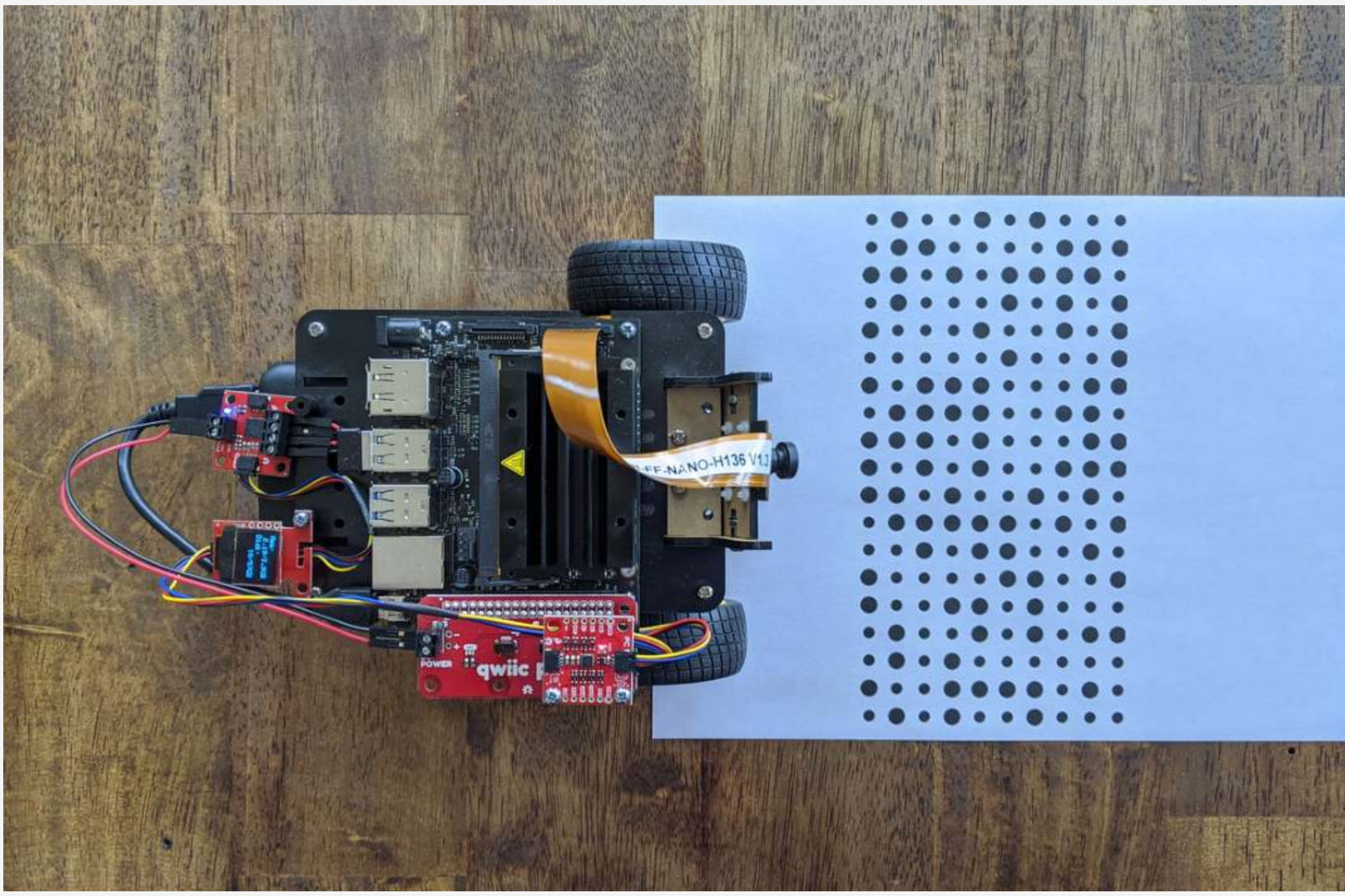
21 / Projective Compensation

26 / Coordinate Frames For Multi-Sensor Systems - Part I

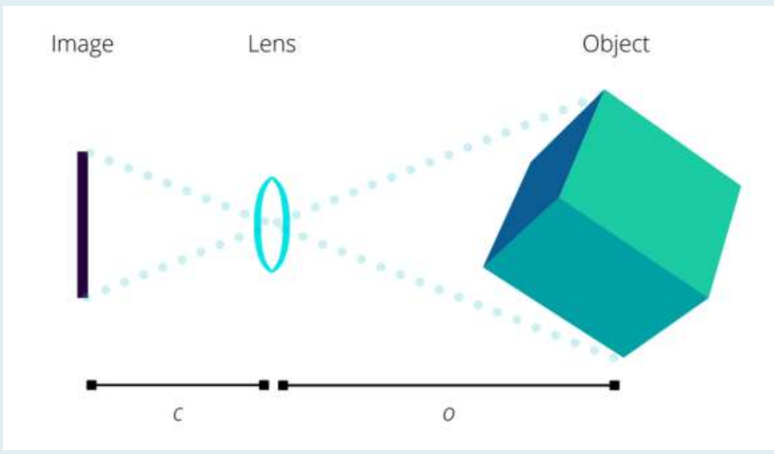
34 / Coordinate Frames For Multi-Sensor Systems - Part II

39 / Kalman Filters for Multi-Sensor Systems - Part I

43 / Kalman Filters for Multi-Sensor Systems - Part II



CAMERA MODELING: FOCAL LENGTH & COLINEARITY



Camera calibration is a deep topic that can be pretty hard to get into. While the geometry of how a single camera system works and the physics describing lens behavior is well-known, camera calibration continues to be mystifying to many engineers in the field.

To the credit of these many engineers, this is largely due to a wide breadth of jargon, vocabulary, and notation that varies across a bunch of different fields. Computer vision seems as if it is a distinct field unto itself, but in reality it was borne from the coalescence of computer science, photogrammetry, physics, and artificial intelligence. As a result of this, the language we use has been adopted from several different disciplines and there's a lot of cross-talk.

In this chapter, we want to explore one of the fundamental aspects of the calibration problem: choosing a model. More specifically, we're going to take a look at how we model cameras mathematically, and in particular take a look at existing models in use today and some of the different approaches they take. For this first essay, I want to specifically take a look at modeling the focal length, and why the common practice of modeling f_x and f_y is sub-optimal when calibrating cameras

Background

To ensure you understand this chapter, we recommend reading the "Building Calibration From Scratch" and "Coordinate Frames For Multi-Sensor Systems Part II" chapters. These will provide a foundation to define the collinearity function. Beyond the above, having some least-squares or optimization knowledge will be helpful, but not necessary.

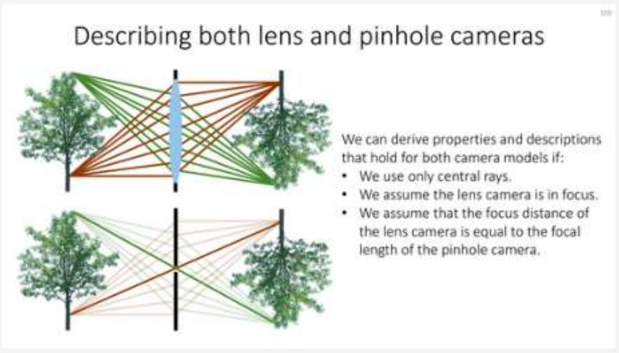
Let's first start by demonstrating the geometry of the problem. In the image at the top of this page, we show the geometry of how a camera captures an image of an object through a lens following the pinhole projection model. On the left-hand side we have the image plane. In analogue cameras, this was a strip of photosensitive film stretched across a backplate behind the lens, or a glass plate covered in photosensitive residue. In modern digital cameras, our image "plane" is an array of pixels on a charge-coupled device (CCD).

On the right we have the object we're imaging. The pinhole-projection model of our lens creates two similar triangles in between these. Effectively, for every ray of light that bounces off of the object, there is a straight ray that can be traced back through the lens center (our pinhole) and directly onto the image plane.

There are two listed distances in this diagram, namely *o* and *c*:

- *o* is the distance of our object at the focus point
- *c* is what we refer to as the *principal distance*

These distances relate to the lens model that we typically use when doing computer vision or photogrammetry. While the collinearity function that we're going to define is going to be based on a pinhole-projection model, we have to also consider the physics of our lens. Carnegie-Mellon University actually has a really good graphic for demonstrating how and when we can relate our lens geometry to the pinhole-projection model:


Describing both lens and pinhole cameras

We can derive properties and descriptions that hold for both camera models if:
- We use only central rays.
- We assume the lens camera is in focus.
- We assume that the focus distance of the lens camera is equal to the focal length of the pinhole camera.

Fig 1: Slide demonstrating the relationship and assumptions relating the pinhole-projection model to the lens model of our camera. © Ioannis Gkioulekas, CMU, 2020.

Often times there is little distinction made between the principal distance *c* and the focal length f of the camera. This is somewhat of an artifact of the fact that most photogrammetric systems were focused at infinity. If we take the lens equation for a thin convex lens:

$$\frac{1}{f} = \frac{1}{o} + \frac{1}{c}$$

We can see how these values are related. As *o* approaches infinity, $\frac{1}{o}$ approaches zero, which makes the focal length and principal distance the same. Note that if we are calibrating / modeling a camera that is not focused at infinity we can still represent our equations with a single focal length, but we need to be careful about collecting measurements from the image, and ensure that we are only collecting measurements that are in focus.

We're going to take some liberties here regarding focal length / principal distance. With the collinearity function we're going to define, we are exclusively measuring principal distance (the distance between the lens and the image plane). Focal length is different from the principal distance, but the ways in which it can deviate from principal distance are separate from the discussion here regarding f_x and f_y as model parameters.

Likewise, for the sake of what we're discussing here we're going to assume that there is no distortion (i.e. that the dotted lines in our first figure are perfectly straight). Distortion doesn't change any of the characteristics we're discussing here with respect to our focal length as it is a different modeling problem.

Collinearity Condition

The graphic above demonstrates the primary relationship that is used when understanding the geometry of how images are formed. In effect, we represent the process of taking an image as a projective coordinate transformation between two coordinate systems. If we were to write this out mathematically, then we would write it as we would any 3D coordinate transformation:

$$\begin{bmatrix} x_i \\ y_i \\ f \end{bmatrix} = \Gamma_o^i \begin{bmatrix} X_o \\ Y_o \\ Z_o \end{bmatrix}$$

Where *o* as a sub/superscript denotes the object space coordinate frame, and *i* is a sub/superscript denotes the image plane. In essence, Γ_o^i is a function that moves a point in object space (the 3D world) to image space (the 2D plane of our image). Note that f in the above equation is constant for all points, because all points lie in the same image plane.

For the sake of simplicity, let's assume for this modeling exercise that Γ_o^i is known and constant. This lets us substitute the right-hand side of the equation with:

$$\begin{bmatrix} X_t \\ Y_t \\ Z_t \end{bmatrix} = \Gamma_o^i \begin{bmatrix} X_o \\ Y_o \\ Z_o \end{bmatrix}$$

Where subscript t denotes the transformed point. We can re-order these equations to directly solve for our image coordinates:

$$\begin{bmatrix} x_i/f \\ y_i/f \\ 1 \end{bmatrix} = \begin{bmatrix} X_t/Z_t \\ Y_t/Z_t \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} f \cdot X_t/Z_t \\ f \cdot Y_t/Z_t \end{bmatrix}$$

Remember, this is just a standard projective transformation. The focal length f has to be the same for every point, because every point lies within our ideal plane. The question remains: how do we get f_x and f_y as they are traditionally modeled? Well, let's try identifying what these were proposed to fix, and evaluate how that affects our final model.

Arguments of Scale

The focal length f helps define a measure of scale between our object-space coordinates and our image-space coordinates. In other words, it is transforming our real coordinates in object space (X_t, Y_t, Z_t) into pixel coordinates in image space (x_i, y_i) through scaling. How we define our focal length has to be based in something, so lets look at some arguments for how to model it.

Arguments of Geometry

From our pinhole projection model, we are projecting light rays from the center of the lens (our principal point) to the image plane. From a geometric or physical perspective, can we justify both f_x and f_y ?

The answer is **NO**, we cannot. The orthonormal distance of the principal point to the image plane is always the same. We know this because for any point *q* not contained within some plane *B*, there can only ever be one vector (line) *v* that is orthogonal to *B* and intersects *q*.

From that perspective, there is ***no geometric basis for having two focal lengths***, because our projective scale is proportional to the distance between the image plane and principal point. There is no way that we can have two different scales! Therefore, two focal lengths doesn't make any sense.

Here, I am avoiding the extensive proof for why there exists a unique solution for *v* intersecting *B* where $v \perp B$ and $\forall q \in B : v \neq q$.

If you're really interested, I suggest Linear and Geometric Algebra by Alan Macdonald, it does contain such a proof in its exercises. If you're interested in how point to plane distance is calculated, see this excellent tutorial.

Argument of History

Okay, so perhaps geometrically this lens model is wrong. However, we don't live in a perfect world and our ideal physics equations don't always model everything. Are f_x and f_y useful for some other historical reason? Why did computer vision adopt this model in the first place?

Well, the prevailing mythos is that it has to do with those darn rectangular pixels! See this StackOverflow post that tries to justify it. This preconception has done considerable damage by perpetuating this model. Rather than place blame though, lets try to understand where this misconception comes from. Consider an array of pixels:

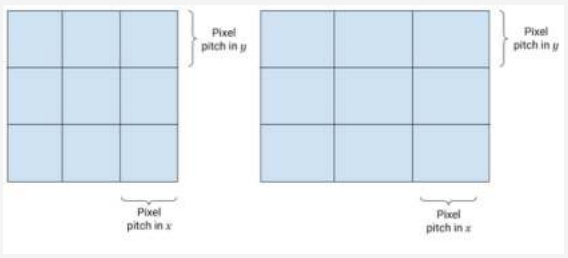


Fig. 2: Two possible arrays of pixels. On the left, we have an ideal pixel grid, where the pixel pitch in *x* and *y* is the same (i.e. we have a square pixel). On the right, we have a pixel grid where we do not have equal pitch in *x* and *y* (i.e. we have rectangular pixels).

The reason that f_x and f_y are used is to try and compensate for the effect of rectangular pixels, as seen in the above figure. From our previous equations, we might model this as two different scales for our pixel sizes in the image plane:

$$\begin{bmatrix} x_i \cdot \rho_x \\ y_i \cdot \rho_y \end{bmatrix} = \begin{bmatrix} f \cdot X_t/Z_t \\ f \cdot Y_t/Z_t \end{bmatrix}$$

Where ρ_x is the pixel pitch (scale) of pixels in the *x*-direction, and ρ_y is the pixel pitch (scale) of pixels in the *y*-direction. Eventually we want to move these to the other side as we did before, which gives us:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} \frac{f}{\rho_x} \cdot X_t/Z_t \\ \frac{f}{\rho_y} \cdot Y_t/Z_t \end{bmatrix}$$

If we then define:

$$f_x = f/\rho_x$$
$$f_y = f/\rho_y$$

Then we have the standard computer vision model of the collinearity equation. From the perspective of what's going one with this example, it isn't completely wild to bundle these together. Any least-squares optimization process would struggle to estimate the focal length and pixel pitch as separate quantities for each dimension. From that perspective, it seems practical to bundle these effects together into f_x and f_y terms. However, now we've tried to conflate our model between something that does exist (different pixel sizes) with something that does not exist (separate distances from the image plane in the *x*- and *y*-directions).

But this ***assumes*** that pitch is applied as a simple multiplicative factor in this way. Modeling it this way does not typically bode well for the adjustment. The immediate question is then: is there a better way to model this? Can we keep the geometric model and still account for this difference? The answer, unsurprisingly, is yes. However, to get there we first have to talk more about this scale difference and how we might model it directly in the image plane.

Differences in Pixel Pitch

Whenever adding a new term to our model, that term must describe an effect that is observable (compared to the precision of our measurements in practice). It doesn't make sense to add a term if it doesn't have an observable effect, as doing so would be a form of *over-fitting*. This is a waste of time, sure, but it also risks introducing *new* errors into our system, which is not ideal.

So the first questions we have to ask when thinking about pixel pitch is: how big is the effect, and how well will we able to observe it? This effect appears as if there is a scale difference in *x* and *y*. This could be caused by one of a few factors:

- Measured physical differences due to rectangular pixels.
- Differences between the clock frequencies of the analogue CCD / CMOS clock and sampling frequency of the Digital-Analog Converter (DAC) in the frame grabber of the camera hardware.
- The calibration data set has or had an insufficient range of Z_o values in the set of object-space points. Put more directly, this is a lack of depth-of-field in our object points. Often this occurs when calibration targets are within a single plane (i.e. when using just a checkerboard, and all your points are in a single plane).

Measured Physical Differences

We've already discussed what physical differences in the pixel size looks like (i.e. rectangular pixels). In general, this is fairly rare. Not because rectangular pixels don't exist, but because measuring each pixel is fairly difficult: pixels are really small. For some cameras, this isn't as difficult; many time-of-flight cameras, for example, can have pixels with a pitch of approximately 40µm (a relatively large size). But for many standard digital cameras, pixel sizes range from 3µm - 7µm.

If we're trying to estimate differences between the pitch in *x* and *y*, how large do they have to be before we notice them? If the difference was approximately 10% of a pixel, then for a 3µm pixel we'd need to be able to observe changes in the image plane of approximately 0.3µm. This is minuscule. The difference in pixel size would have to be much larger in order for us to actually observe such a difference.

Clock-Frequency Differences

Scale differences can also "appear" out of our data due to frequency differences between the frequency of the CCD pixel clock and the sampling frequency of the DAC in the frame grabber of the digital camera. For a single row of pixels, this frequency difference would look similar to:

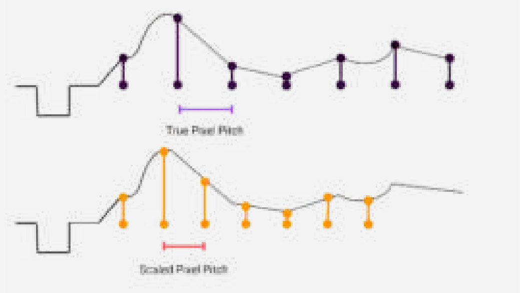


Fig. 3: Example of a signal being read out along a line of pixels. Each vertical colored line is a pixel. In the top version, the CCD pixel clock is what determines the frequency of readout. In the bottom, the DAC sampling frequency is used, which produces a perceived difference in the final pixel pitch.

The purple lines are the true locations in the signal that we should be sampling, but the orange lines are where we actually sample. We can see that the sampled pixels (in orange) have a different pitch. The columns (not shown here) are sampled evenly at the original pixel pitch, but the rows were sampled incorrectly at the scaled pixel pitch. As a result, we would observe two different scales in the *x* and *y* directions.

This effect can be overcome today with pixel-synchronous frame grabbing (e.g. in a global shutter synchronous camera) or on board digital-analog conversion. However, this mechanic isn't in every camera, especially when it comes to cheap off-the-shelf cameras.

It may also exist due to the CCD array configuration in hardware, if pixels have different spacing in *x* and *y*, but that again goes back to the first point where we have to be able to measure these physical differences somehow. Nonetheless, the effect is the same on the end result (our pixel measurements).

The above "array configuration" problem is common when for example we "interpolate" our pixel data from a Bayer pattern of multiple pixels. This makes the spacing uneven as we often have more of one color pixel (green) than we do of other kinds of pixels (red, blue). See our guest post on OpenCV's blog for more info.

Insufficient Depth-of-Field

Even if we assume that our hardware was perfect and ideal, and that our pixels were perfectly square, we might still observe some scale difference in *x* and *y*. This could be because our calibration data set comprises only points in a single plane, and we lack any depth-of-field within the transformation.

Unfortunately, this ends up appearing as a type of *projective compensation* as a result of the first derivative of our ideal model. Projective compensation can occur when two parameters (e.g. in a calibration) influence each other's values because they cannot be estimated independently.

This is the point where we would be going off into the weeds of limits and derivatives, so we will leave the explanation at that for now. The important consideration is that even in an ideal world, we might still actually observe this effect, and so we should model it!

Scale and Shear in Film

So we've now ascertained that modeling these scale differences as a factor of f_x and f_y doesn't make sense geometrically. However, that doesn't mean that we aren't going to see this in practice, even if our camera is close to perfect. So how are we to model this scale effect? Fortunately, the past has an answer! Scale (and shear) effects were very common back when cameras used to be analogue and based on physical film.

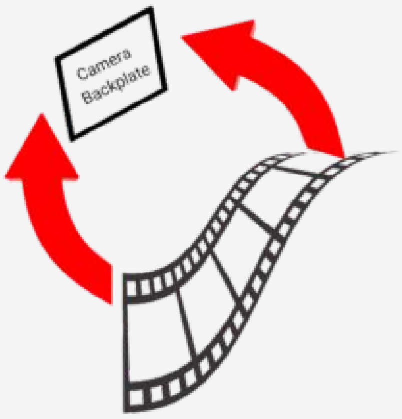


Fig. 5: Example of how film is stretched across a backplate in a traditional analogue camera.

Consider the above scenario where we are stretching a film of some material across a backplate in the camera. Tension across the film being stretched over the backplate was in the *x* direction, but not the *y* direction. This difference in tension could cause a scale difference in the final developed image, as that tension is not present when the film is developed.

Back then, shear effects were also modeled in addition to scale. This effectively meant that the tension along one diagonal of the image was not equal to the tension along the other diagonal. This would result in a warping or shear effect in the final image plane. Nowadays, this would mean that our pixels are not a rectangle or a square, but rather some rhombus. This isn't particularly realistic for modern digital cameras, so we ignore it here.

In any case, we end up with the following additional parameter to account for scale differences.

$$\begin{bmatrix} x_i - a_1 x_i \\ y_i \end{bmatrix} = \begin{bmatrix} f \cdot X_t / Z_t \\ f \cdot Y_t / Z_t \end{bmatrix}$$

Where a_1 is a multiplicative factor applied to the *x* coordinate. This model has a foundation in the literature dating back decades.

Is There A Difference Between These Two Models?

Mathematically, yes! This different model has a couple things going for it:

- We maintain the original projective geometry of the collinearity function.
- We model scale differences in *x* and *y* in the image plane directly.
- Our calibration process has better observability between f and a_1 because we aren't conflating different error terms together and are instead modeling them independently based off a physical model of how our system behaves.

That last point is the most important. Remember the original systems of equations we ended up with:

$$\frac{x_i}{f} = \frac{X_t}{Z_t}$$

$$\frac{y_i}{f} = \frac{Y_t}{Z_t}$$

From this, we bundled together the pixel pitch as if that was the only effect. This was possible because when optimizing, it's not possible to observe differences between two multiplicative factors, e.g.:

$$w = r \cdot s \cdot v$$

If we know *v* and *w* as say, 2 and 8 respectively, there's no way for us to cleanly estimate *r* and *s* as separate quantities. Pairs of numbers that would make that equality true could be 2 and 2, or 4 and 1, or any other multitude of factors. In the same way, if we try to reconstruct our original definitions for f_x and f_y we would get:

$$x_i - a_1 x_i = f(X_t/Z_t)$$
$$x_i(1 - a_1) = f(X_t/Z_t)$$
$$f_x = f/(1 - a_1)$$

and similarly for *y*, where we would get:

$$f_y = f/(1 - 0)$$

However, remember that this only works if scale differences are the only error factor in our image plane. This is not often the case. If we have additional errors ϵ_x and ϵ_y in our system, then instead we would get:

$$f_x = f/(1 + a_1 + \frac{\epsilon_x}{x_i})$$

$$f_y = f/(1 + \frac{\epsilon_y}{y_i})$$

Rather than modeling our calibration around the physical reality of our camera systems, this bundles all of our errors into our focal length terms. This leads to an instability in our solution for f_x and f_y when we perform a camera calibration, as we end up correlating scene or image-specific errors of the data we collected (that contain $1/x_i$ or $1/y_i$) into our solution for the focal length. If we don't use f_x and f_y, and ϵ_x and ϵ_y are uncorrelated (i.e. independent variables) then we can easily observe f independently. **In doing so, we have a stronger solution for our focal length that will generalize beyond just the data set that was used in the calibration to estimate it.**

Now obviously, you would say, we want to model ϵ_x and ϵ_y, so that these are calibrated for in our adjustment. Modeling our system to match the physical reality of our camera geometry in a precise way is one key to removing projective compensation between our parameters and to estimate all of our model parameters independently. By doing so, we avoid solving for "unstable" values of f_x and f_y, and instead have a model that generalizes beyond the data we used to estimate it.

At the end of the day, it is easier to get a precise estimate of:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} f \cdot X_t/Z_t \\ f \cdot Y_t/Z_t \end{bmatrix} + \begin{bmatrix} a_1 x_i \\ 0 \end{bmatrix}$$

as a model, over:

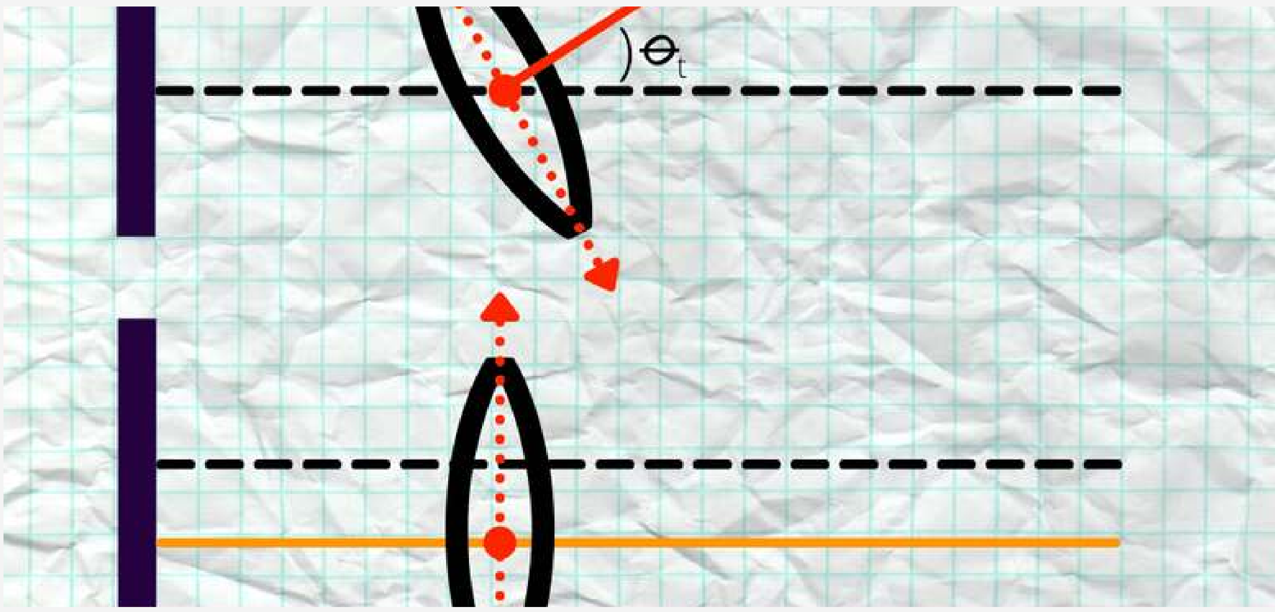
$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} f_x \cdot X_t/Z_t \\ f_y \cdot Y_t/Z_t \end{bmatrix}$$

Conclusion

Hopefully we've demonstrated why the classical use of f_x and f_y in most calibration pipelines is the wrong model to pick. There's always different abstractions for representing data, but using a single focal length term better matches the geometry of our problem, and doesn't conflate errors in the image plane (scale differences in *x* and *y* directions of that plane) with parameters that describe geometry outside that image plane (the focal distance). Most importantly, a single focal length avoids projective compensation between the solution of that focal length and measurement errors in our calibration data set.

Modeling sensors is a difficult endeavor, but a necessary step for any perception system. Part of what that means is thinking extremely deeply about how to model our systems, and how to do so in the way that best generalizes across every scene, environment, and application. While we've given a bit of a taste of what that looks like here for cameras, this may be a lot if you just want your system to work. For a shortcut, try the Tangram Vision Platform if you want to stop worrying about camera models, and start building the next generation of perception-based technology today!

CAMERA MODELING: EXPLORING DISTORTION AND DISTORTION MODELS



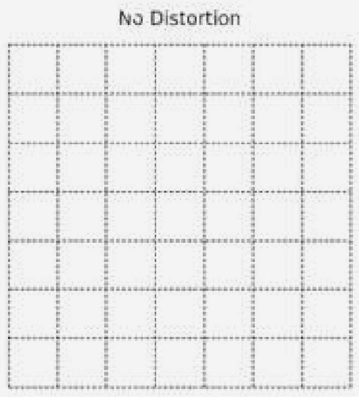
In this chapter, we want to explore another part of the camera modeling process: modeling lens distortions. One assumption behind our pinhole-projection model is that light travels in straight rays, and does not bend. Of course, our camera's lens is not a perfect pinhole and light rays will bend and distort due to the lens' shape, or due to refraction.

What does lens distortion look like?

Before we get into the actual models themselves, it is good to first understand what lens distortion looks like. There are several distinct types of distortions that can occur due to the lens or imaging setup, each with unique profiles. In total, these are typically broken into the following categories:

- Symmetric Radial Distortion
- Asymmetric Radial Distortion
- Tangential or De-centering Distortion

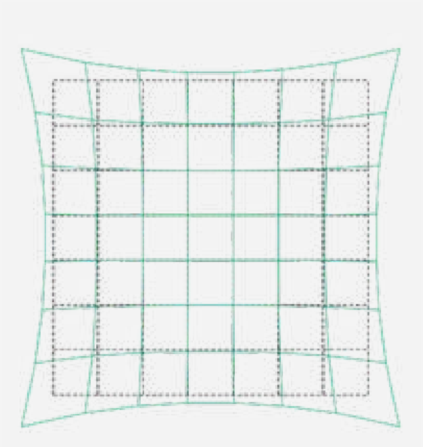
Each of these is explored independently below. For each of these distortions, consider the base-case where there is no distortion:



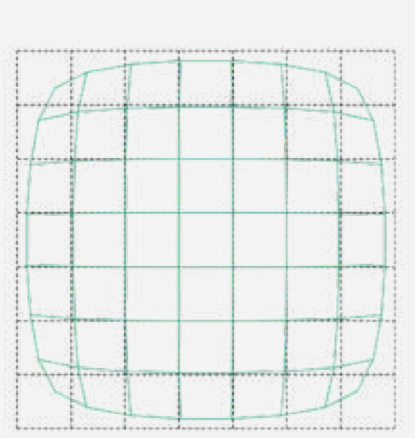
In the above figure, we represent the image plane as a grid. As we discuss the different kinds of distortions, we'll demonstrate how this grid is warped and transformed. Likewise, for any point (x, y) that we refer to throughout this text, we're referring to the coordinate frame of the image plane, centered (with origin) at the principal point. This makes the math somewhat easier to read, and doesn't require us to worry about column and row offsets c_x and c_y when trying to understand the math.

Symmetric Radial Distortions

Symmetric radial distortions are what are typically imagined when discussing image distortion. Often, this type of distortion will be characterized depending on if it is positive (pincushion) or negative (barrel) distortion.



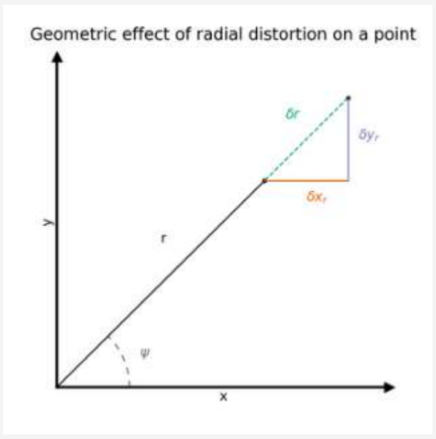
Positive (Pincushion) Radial Distortion
It is "positive" because the projected distance increases beyond the expected locations as one moves farther away from the center of the image plane.



Negative (Barrel) Radial Distortion
It is "negative" because the projected distance decreases from the expected locations as one moves farther away from the center of the image plane

While the two distortions might seem as if they are fundamentally different, they are in fact quite alike! The amount of distortion in the teal lines is greater in magnitude at the edges of our image plane, while being smaller in the middle. This is why the black and teal lines overlap near the centre, but soon diverge as distance increases.

Symmetric radial distortion is an after-effect of our lens not being a perfect pinhole. As light enters the lens outside of the perspective centre it bends towards the image plane. It might be easiest to think of symmetric radial distortion as if we were mapping the image plane to the convexity or concavity of the lens itself. In a perfect pinhole camera, there wouldn't be any distortion, because all light would pass through a single point!



Geometric effect of radial distortion applied to a point in our image plane coordinate frame. The effect is characterized along the radial direction.

This distortion is characterized as symmetric because it only models distortion as a function of distance from the centre of the image plane. The geometric effect of radial distortion is only in the radial direction, as characterized by δr in the above figure.

Asymmetric Radial Distortions

Asymmetric radial distortions are radial distortion effects much like the above, but unlike symmetric radial distortion, asymmetric radial distortion characterizes distortion effects that are dependent both on the distance from the image centre as well as how far away the object being imaged is. Asymmetric radial effects are most pronounced in two scenarios:

1. Cameras with long focal lengths and very-short relative object distances. e.g. a very-near-field telephoto lens that is capturing many objects very close.
2. Observing objects through a medium of high-refraction, or differing refractive indices. e.g. two objects underwater where one is near and one is far away.

This type of distortion is typically tricky to visualize, as well as to quantify, because it is dependent on the environment. In most robotic and automated vehicle contexts, asymmetric radial distortion is not a great concern! Why? Well, the difference in distortions depends on the difference in distances between objects. This is usually because of some kind of refractive difference between two objects being imaged, or because the objects are out of focus of the camera (i.e. focal length is too large relative to the object distance).

Neither of the above two scenarios are typical; as such, asymmetric radial distortion is an important aspect of modeling the calibration in applications when these scenarios are encountered.

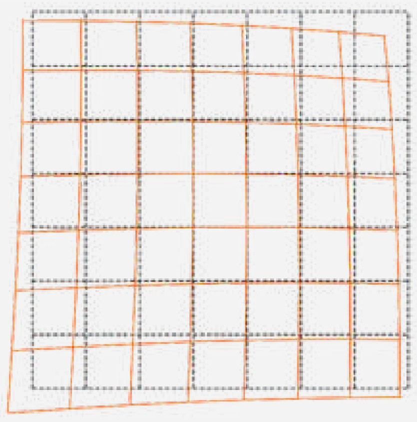
In most robotic contexts, the primary use for imaging and visual-odometry is done in relatively short ranges with cameras that have short focal lengths, and the primary medium for light to travel through is air.

Since there doesn't tend to be big atmospheric variances between objects that are close, and since light is all traveling through the same medium, there isn't much of an asymmetric refractive effect to characterize or measure. As a result, this kind of radial distortion isn't common when calibrating cameras for these kinds of applications. If we can't measure it, we shouldn't try to model it!

You may be left wondering what is meant by "close" when discussing "atmospheric variance between objects that are close." In short: it is all relative to the refractive index of the medium in which you are imaging. On the ground it is atypical to have atmospheric effects (we don't see pockets of ozone on the ground, as an example).
This is not generally true underwater, nor if you are performing some kind of aerial observation from the stratosphere or higher. Additionally, even in such scenarios, if all observed objects are roughly the same distance away then it doesn't matter since asymmetric radial distortion characterizes radial effects as a result of the extra distance light travels (and refracts) between two objects at different distances from the camera.

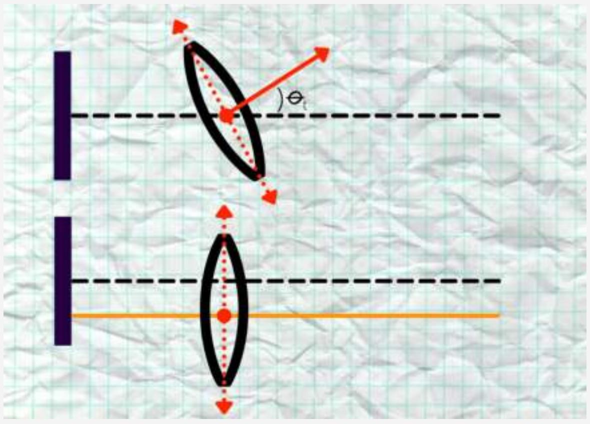
Tangential (De-centering) Distortions

The last kind of distortions to characterize in a calibration are tangential effects, often as a result of de-centered lens assemblies. First, an (exaggerated) example of what these might look like:



Unlike radial distortion, the image plane is skewed, and the distance from center is less important. The effect is exaggerated: most cameras do not have tangential distortions to this degree.

Tangential distortion is sometimes also called de-centering distortion, because the primary cause is due to the lens assembly not being centered over and parallel to the image plane. The geometric effect from tangential distortion is not purely along the radial axis. Instead, as can be seen in the figure above, it can perform a rotation and skew of the image plane that depends on the radius from the image center!

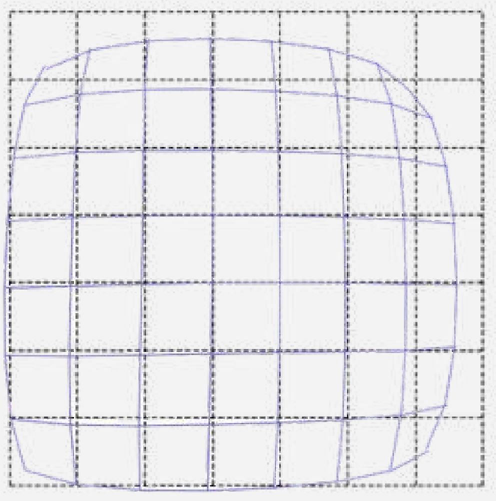


Example of lens geometry as it impacts tangential distortion. In the top graphic, the lens is not parallel with the image plane, which produces tangential distortion. In the lower graphic, the lens is offset from the image plane. While we might normally model this with c_x and c_y for a single lens, if this is one lens offset among a collection of lenses, the final effect projects into our image as tangential or de-centering distortion.

In the above figure, one can see how the lens being either angled with respect to the orthogonal axis of the image plane, or shifted, would project an image into a different spot on the plane. In most cameras used for robotics or automated vehicle applications, tangential distortion will usually be significant enough to model, but is often an order of magnitude smaller than e.g. symmetric radial distortion.

Compound Distortions

Typically when we think of distortion, we try to break down the components into their constituent parts to aid our understanding. However, most lens systems in the real world will have what is often referred to as compound distortion. There's no tricks here, it's simply an aggregate effect of all the previous types of distortions in some combination. This kind of



Example of a compound distortion profile, which combines barrel, pincushion, and tangential effects.

distortion is especially prevalent in cameras with compound lenses, or very complicated lens assemblies.

Common Distortion Models

While there are more models than what is described here, the industry has largely standardized on the following two distortion models.

Brown-Conrady

Brown-Conrady distortion is probably what most think of as the "standard" radial and tangential distortion model. This model finds its roots in two | documents, authored by Brown and Conrady. The documents are quite old and date up to a century ago, but still form the foundation of many of the ideas around characterizing and modeling distortion today!

This model characterizes radial distortion as a series of higher order polynomials:

$$r = \sqrt{x^2 + y^2}$$

$$\delta r = k_1 r^3 + k_2 r^5 + k_3 r^7 + ... + k_n r^{n+2}$$

In practice, only the k_1 through k_3 terms are typically used. For cameras with relatively simple lens assemblies (e.g. only contain one or two lenses in front of the CMOS / CCD sensor), it is often sufficient to just use the k_1 and k_2 terms.

To relate this back to our image coordinate system (i.e. *x* and *y*), we usually need to do some basic trigonometry:

$$\delta x_r = \sin(\psi)\delta r = \frac{x}{r}(k_1 r^3 + k_2 r^5 + k_3 r^7)$$

$$\delta y_r = \cos(\psi)\delta r = \frac{y}{r}(k_1 r^3 + k_2 r^5 + k_3 r^7)$$

The original documents from Brown and Conrady did not express δr in just these terms, and in fact the original documents state everything in terms of "de-centering" distortion broken into radial and tangential components. Symmetric radial distortion as expressed above is a mathematical simplification of the overall power-series describing the radial effects of a lens. The formula we use above is what we call the "Gaussian profile" of Seidel distortion.

Wikipedia has a good summary of the history here, but the actual formalization is beyond the scope of what we want to cover here.

Tangential distortion, as characterized by the Brown-Conrady model, is often simplified into the following *x* and *y* components. We present these here first as they are probably what most are familiar with:

$$\delta x_t = p_1(r^2 + 2x^2) + 2p_2xy$$

$$\delta y_t = p_2(r^2 + 2y^2) + 2p_1xy$$

This actually derives from an even-power series much like the radial distortion is an odd-power series. The full formulation is a solution to the following:

$$\delta t = P(r)cos(\psi - \psi_0)$$

Where *P(r)* is our de-centering distortion profile function, ψ is the polar angle of the image plane coordinate, and ψ_0 is the angle to the axis of maximum tangential distortion (i.e. zero radial distortion). Expanding this into the general parameter set we use today is quite involved (read the original Brown paper!), however this will typically take the form:

$$\delta x_t = [p_1(r^2 + 2x^2) + 2p_2xy](1 + p_3r^2 + p_4r^4 + p_5r^6 + ...)$$

$$\delta y_t = [p_2(r^2 + 2y^2) + 2p_1xy](1 + p_3r^2 + p_4r^4 + p_5r^6 + ...)$$

Because tangential distortion is usually small, we tend to approximate it using only the first two terms. It is rare for de-centering to be so extreme that our tangential distortion requires higher order terms because that would mean that our lens is greatly de-centered relative to our image plane. In most cases, one might ask if their lens should simply be re-attached in a more appropriate manner.

Kannala-Brandt

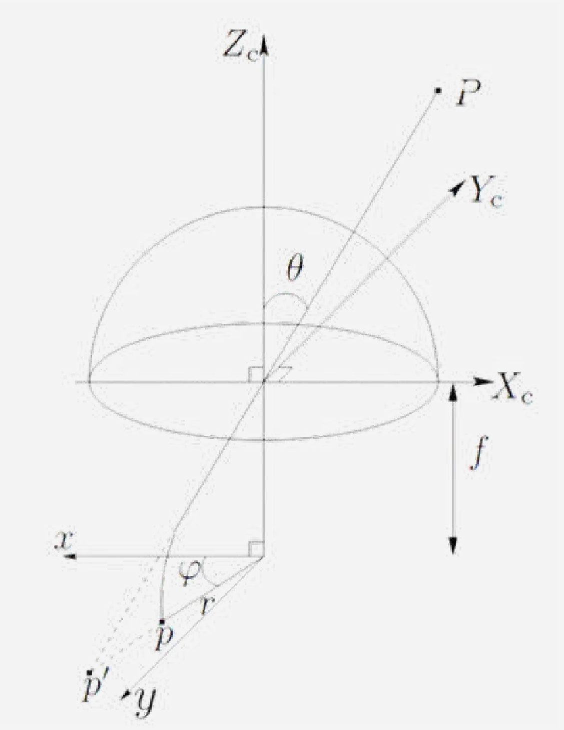
Almost a century later (2006, from the original Conrady paper in 1919), Juho Kannala and Sami Brandt published their own paper on lens distortions. The main contribution of this paper adapts lens distortion modeling to be optimized for wide-angle, ultra wide-angle, and fish-eye lenses. Brown & Conrady's modeling was largely founded on the physics of Seidel aberrations, which were first formulated around 1867 for standard lens physics of the time, which did not include ultra wide and fish-eye lenses.

The primary difference that most folks will notice using this model lies in symmetric radial distortion. Rather than characterizing radial distortion in terms of how far a point is from the image centre (the radius), Kannala-Brandt characterizes distortion as a function of the incidence angle of the light passing through the lens. This is done because the distortion function is smoother when parameterized with respect to this angle θ, which makes it easier to model as a power-series:

$$\theta = \arctan(\frac{r}{f})$$

$$\delta r = k_1\theta + k_2\theta^3 + k_3\theta^5 + k_4\theta^7 + ... + k_n\theta^{n+1}$$

Above, we've shown the formula for θ when using perspective projection, but the main advantage of the Kannala-Brandt model is that it can support different kinds of projection by swapping our formula for θ, which is what makes the distortion function smoother for wide-angle lenses. See the following figure, shared here from the original paper, for a better geometric description of θ:



Projection of quantities used in Kannala-Brandt distortion. Shared from the original paper, "*A Generic Camera Model and Calibration Method for Conventional, Wide-angle, and Fish-eye Lenses*," by J. Kannala and S. Brandt

Kannala-Brandt also aims to characterize other radial (such as asymmetric) and tangential distortions. This is done with the following additional parameter sets:

$$\delta r_{other} = (l_1\theta + l_2\theta^3 + l_3\theta^5)(i_1\cos(\psi) + i_2\sin(\psi) + i_3\cos(2\psi) + i_4\sin(2\psi) + \ldots)$$

$$\delta t = (m_1\theta + m_2\theta^3 + m_3\theta^5)(j_1\cos(\psi) + j_2\sin(\psi) + j_3\cos(2\psi) + j_4\sin(2\psi) + \ldots)$$

Overall, this results in a 23 parameter model! This is admittedly overkill, and the original paper claims as much. These models, unlike the symmetric radial distortion, are an empirical model derived by fitting an N-term Fourier series to the data being calibrated. This is one way of characterizing it, but over-parameterizing our final model can lead to poor repeatability of our final estimated parameters. In practice, most systems will characterize Kannala-Brandt distortions purely in terms of the symmetric radial distortion, as that distortion is significantly larger in magnitude and will be the leading kind of distortion in wider-angle lenses.

Overall, Kannala-Brandt has been chosen in many applications for how well it performs on wide-angle lens types. It is one of the first distortion models to successfully displace the Brown-Conrady model after decades.

Gaussian vs. Balanced Profiles

An astute reader of this article might be thinking right now: "*Hey, <library-that-I-use> doesn't use these exact models! That math seems off!*" You would mostly be correct there: this math is a bit different from how many popular libraries will model distortion (e.g. OpenCV).

Using Brown-Conrady as an example, you might see symmetric radial distortion formulated as so:

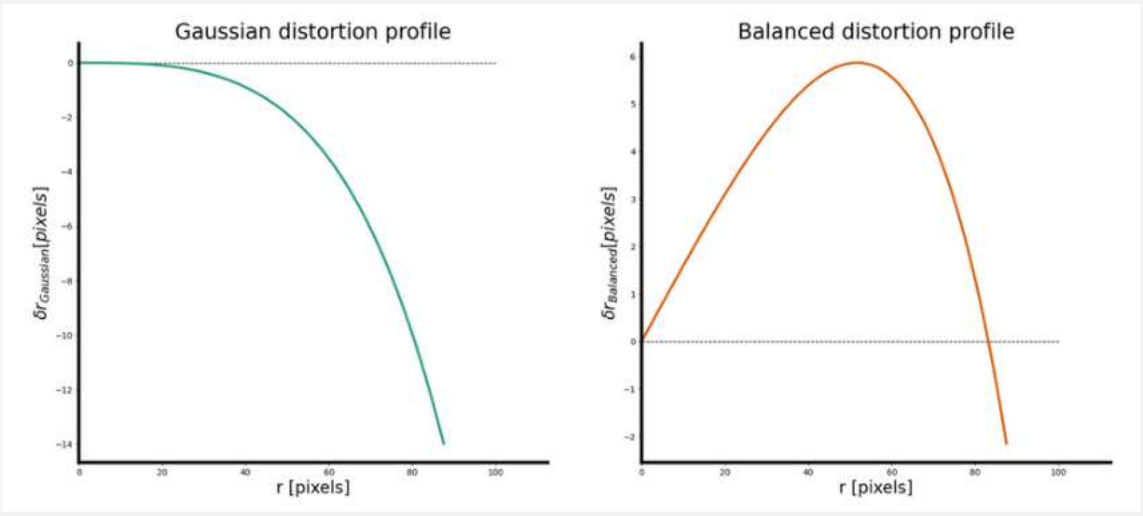
$$\delta r = r + k_1 r^3 + k_2 r^5 + k_3 r^7$$

or for Kannala-Brandt:

$$\theta = \arctan(r)$$

$$\delta r = \theta + k_1\theta^3 + k_2\theta^5 + k_3\theta^7 + k_4\theta^9$$

This probably seems quite confusing, because all these formulae are a bit different from the papers presented in this article. The main difference here is that the distortions have been re-characterized using what is referred to in photogrammetry as the ***Balanced Distortion Profile***. Up until now, we have been presenting distortions using what is referred to in photogrammetry as the ***Gaussian Distortion Profile***.

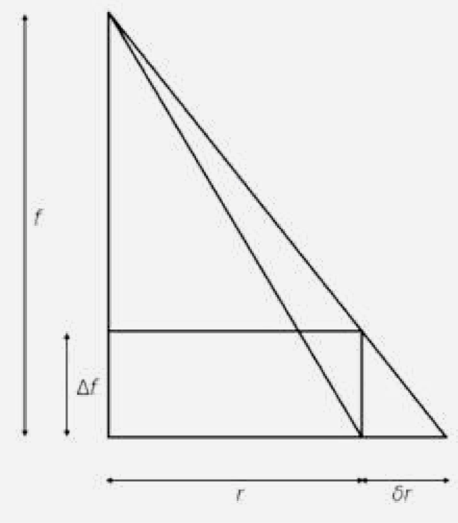


Example comparing two equivalent distortion models. The left describes the Gaussian distortion profile, whereas the right hand side describes a Balanced distortion profile. The distortions are effectively the same, but the model is transformed in the balanced profile scenario. Notice the two different scales in the left and right figures.

As can be seen in the figure above, the profile and maximum magnitude of distortion are fundamentally different in the above cases. More specifically, the balanced profile is one way to limit the maximum distortion of the final distortion profile, while still correcting for the same effects. So how does one go from one representation to the other? For Brown-Conrady, this is done by scaling the distortion profile by a linear correction:

$$\delta r_{Balanced} = k_0' r + k_1' r^3 + k_2' r^5 + k_3' r^7$$

$$\delta r_{Balanced} = \frac{\Delta f}{f} r + \left(1 + \frac{\Delta f}{f}\right)(k_1 r^3 + k_2 r^5 + k_3 r^7)$$



Re-balancing Correction for Radial Distortion
A set of similar triangles representing the linear correction done to re-balance the Gaussian distortion profile. Re-balancing is used to set the distortion to zero at some radius, or to reconfigure the distortion profile to characterize itself for a virtual focal length.

Given that, why does this discrepancy exist? There are a few reasons this could be used. Historically, the balanced profile was used because distortion was measured manually through the use of mechanical stereoplotters. These stereoplotters had a maximum range in which they could move mechanically, so limiting the maximum value of distortion made practical sense.

But what does that mean for today? Nowadays with the abundance of computing resources it is atypical to use mechanical stereoplotters in favor of making digital measurements on digital imagery. There doesn't seem to be a paper trail for why this decision was made, so it could just be a historical artifact. However, doing the re-balancing has some advantages:

- By parameterizing the model with $\frac{\Delta f}{f}$ = 1, as OpenCV does, it makes the math and partial derivatives for self-calibration somewhat easier to implement. This is especially true when using the Kannala-Brandt model, since doing this removes the focal length term from θ = *arctan (r)*, which means that there is one less partial derivative to compute.

- If you keep your entire self-calibration pipeline in units of "pixels" as opposed to metric units like millimeters or micrometers, then the Gaussian profile will produce values for k_1, k_2, k_3 that are relatively small. On systems that do not have full IEEE-754 floats (specifically, systems that use 16-bit floats or do not support 64-bit floats at all), this could lead to a loss in precision. Some CPU architectures today do lack full IEEE-754 support (armel, I'm looking at you), so the re-balancing could have been a consideration for retaining machine precision without adding any specialized code. There is no difference in the final geometric precision or accuracy of the self-calibration process as a result of the re-balancing, as it is just a different model.

- All of the above math is derived from basic lens geometry, and presents a problem if one has to choose a focal length f between f_x and f_y ! So if you're already using f_x and f_y , removing all focal length terms from e.g. Kannala-Brandt distortions may appear to save you from having to make poor approximations of the quantities involved.

Despite these reasons, there is also one very important disadvantage. Notably: the balanced profile with Brown-Conrady distortions introduces a factor of $(1 + \frac{\Delta f}{f})$ into the determination of k_1 through k_3 . This may not seem like much, but it means that we are introducing a correlation in the determination of these parameters with our focal length. If one chooses a model where f_x and f_y are used, then the choice in parameterization will make the entire calibration unstable, as small errors in the determination of any of these parameters will bleed into every other parameter. This is one kind of projective compensation, and is another reason for why the blog post we wrote on the subject suggested not to use this parameterization.

It might also seem that with the Kannala-Brandt distortion model, we simplify the math by cancelling out f from our determination of θ. This is true, and it will make the math easier, and remove a focal length term from our determination of the Kannala-Brandt parameters. However, if one chooses a different distortion projection, e.g. orthographic for a fish-eye lens:

$$\theta_{Gaussian} = \arcsin\left(\frac{r}{f}\right)$$

$$\theta_{Balanced} = \arcsin(r)$$

Then one will quickly notice that the limit of $\theta_{Balanced}$ is not well defined as $r \rightarrow \infty$, and gives us a value of $(-i)\infty$. As we've tried to separate our focal length from our parameters, we've ended up wading into the territory of complex numbers! Since our max radius and focal length are often proportional, $\theta_{Gaussian}$ does not suffer from the same breakdown in values at the extremes.

It may seem like Kannala-Brandt is a poor choice of model as a result of this. For lenses with a standard field-of-view, the Brown-Conrady model with the Gaussian profile does a better job of determining the distortion without introducing data correlations between the focal length and distortion parameters.

However, the Brown-Conrady model does not accommodate distortions from wide-field-of-view lenses very well, such as when using e.g. fisheye lenses. This is because they were formulated on the basis of Seidel distortions, which do not operate the same way as the field-of-view increases past 90°.

The Kannala-Brandt model, while introducing some correlation between our determination of f and our distortion coefficients k_1 through k_4 , does a better job of mapping distortion with stereographic, orthographic, equidistance, and equisolid projections. As with anything in engineering there are trade-offs, and despite the extra correlations, the Kannala-Brandt model will still often provide better geometric precision of the determined parameters compared to the Brown-Conrady model in many of these scenarios.

As can be seen, chasing simplicity in the mathematical representations is one way in which our choice of model can result in unstable calibrations, or nonsensical math. Given that we want to provide the most stable and precise calibrations possible, we lean towards favouring the Gaussian profile models where possible. It does mean some extra work to make sure the math is correct, but also means that by getting this right once, we can provide the most stable and precise calibrations ever after.

Wrapping Up

We've explored camera distortions, some common models for camera distortions, and explored the ways in which these models have evolved and been implemented over the past century. The physics of optics has a rich history, and there's a lot of complexity to consider when picking a model.

Modeling distortion is one important component of the calibration process. By picking a good model, we can reduce projective compensations between our focal length f and our distortion parameters, which leads to numerical stability in our calibration parameters.

Fitting in with our previous camera model, we can formulate this in terms of the collinearity relationship (assuming Brown-Conrady distortions):

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} f \cdot X_t/Z_t \\ f \cdot Y_t/Z_t \end{bmatrix} + \begin{bmatrix} a_1 x_i \\ 0 \end{bmatrix} + \begin{bmatrix} x_i(k_1 r^2 + k_2 r^4 + k_3 r^6) \\ y_i(k_1 r^2 + k_2 r^4 + k_3 r^6) \end{bmatrix} + \begin{bmatrix} p_1(r^2 + 2x_i^2) + 2p_2 x_i y_i \\ p_2(r^2 + 2y_i^2) + 2p_1 x_i y_i \end{bmatrix}$$

or, with Kannala Brandt distortions:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} f \cdot X_t/Z_t \\ f \cdot Y_t/Z_t \end{bmatrix} + \begin{bmatrix} a_1 x_i \\ 0 \end{bmatrix} + \begin{bmatrix} \frac{x_i}{r}(k_1\theta + k_2\theta^3 + k_3\theta^5 + k_4\theta^7) \\ \frac{y_i}{r}(k_1\theta + k_2\theta^3 + k_3\theta^5 + k_4\theta^7) \end{bmatrix}$$

Did we get too far into the weeds with this? Never fear, we have you covered. If you're worried about the best model for your camera and would rather do *anything else*, check out the Tangram Vision Platform. We're building perception tools that lift this and other burdens off your shoulders.

PROJECTIVE COMPENSATION

Projective compensation affects many calibration scenarios. In this chapter, we'll explore what it is, how to detect it, and how to address it.

In previous chapters, we've discussed some of the modeling approaches we take here at Tangram Vision, and tried to explore the history of how these modeling approaches were formed. In "Camera Modeling: Exploring Distortion And Distortion Models", we mentioned the topic of projective compensation. Projective compensation is prevalent across most forms of optimization, but is so rarely discussed in a computer vision context.

We aim to provide both a brief intuition behind projective compensation and examples where it crops up in camera calibration. All optimization processes need to consider this topic to some extent, but it holds particular importance when performing calibration. Limiting the effects of projective compensation is one of the easiest ways to ensure that a calibration is consistent and stable.

What is Projective Compensation?

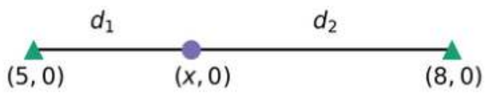
In optimization, process errors in the estimation of one parameter **project** into the solution for other parameters. Likewise, a change in a parameter's solution **compensates** for errors in the solution of other parameters. This process of **projective compensation** can happen for any number of parameter errors and solutions in a system.

Visualizing a 2D Example

Follow along: all of the code used to run the least-squares problems can be found in the Tangram Visions Blog Repository.

Let us first consider a Very Simple Problem: we want to determine the x position of a point q (the circle) located between two known points (the triangles).

Unfortunately, the only way to find x is to measure its position *relative to our known points*. This means that we'll have two distances (d_1 and d_2) that should help us solve our problem.

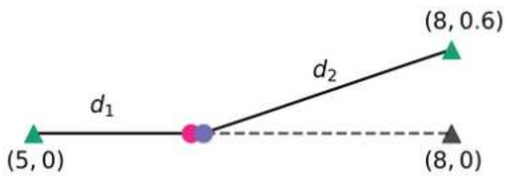


After our measurements, we determine that d_1 = 1.12 and d_2 = 1.86. Since we have more observations (two) than unknowns (one), this problem is *over-determined* and we can use a least-squares optimization to solve. Running this process, we get the following results:

Name	Value	Description
d	[1.12, 1.86]	The array of observations within our system
x	6.13	The solution of our x-coordinate
r	[0.01, 0.01]	Residual error for each of our observations (in a full camera calibration, this might be referred to as "reprojection error")

So far, so good, if not terribly interesting. In fact, using an optimization process for this might be a bit overkill, given that we could have just as easily taken the average from our observations.

Let's add a twist: suppose that when we were recording the positions of our known points (the triangles), we accidentally got the y-coordinate of one of them wrong. Instead of the point being recorded as being at (8, 0), we recorded the point to be at (8, 0.6):



Notice that neither d_1 or d_2 have changed, since those were honest observations written down correctly. However, the point that describes q is no longer as tightly determined. Based on our two observations, q could be in either the position of the pink circle or in the position of the purple circle.

Luckily, our problem is still over-determined (i.e. we have more observations of the point than we do unknowns to solve for), so again we can perform a least-squares optimization:

Name	Value	Description
d	[1.12, 1.86]	The array of observations within our system
x	6.177	The solution of our x-coordinate
r	[0.057, 0.059]	Residual error for each of our observations

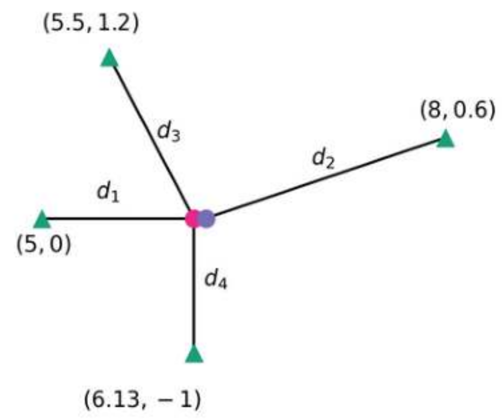
As can be seen, the solution for x has changed dramatically, and the final result has much larger residual error (by a factor of 5-6!).

This is projective compensation at work. Errors in one quantity (the y-coordinate of a control point) are being compensated for by projecting that error across other quantities that are being estimated (the x-coordinate of our point of interest, q). The least-squares optimization still ran, and still minimized error as much as it could, but that error is still in the optimization and has to be accounted for somewhere.

The Effect of Data

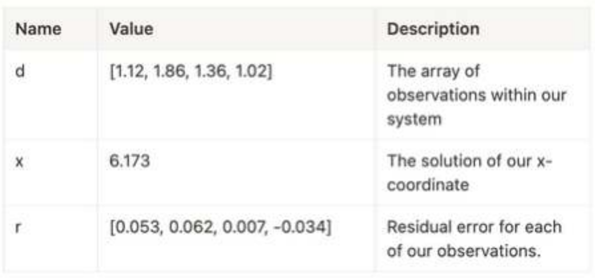
One way to reduce the impact of projective compensation is to have more data.

Let's say we add two more points, as shown in the following figure:



These two new points agree that q is likely to be on the pink point, whereas we still have our erroneous point at (8, 0.6) that would imply q should be at the purple location. How much of an effect does this have?

If we say d_3 = 1.36 and d_4 = 1.02, the least-squares results are then:



Name	Value	Description
d	[1.12, 1.86, 1.36, 1.02]	The array of observations within our system
x	6.173	The solution of our x-coordinate
r	[0.053, 0.062, 0.007, -0.034]	Residual error for each of our observations.

This final value for x (6.173) does start to approach our original true value (6.13), but doesn't quite get there all the way. While additional data does make the situation better, it doesn't entirely ameliorate our problem.

This demonstrates how projective compensation can also be an effect of our underlying model. If we can't optimize the x and y-coordinates of our "known" points, projective compensation will still occur; we just can't do anything about it.

Solving For Projective Compensation

Sadly, the heading above is somewhat of a trick — it is not really possible to "solve for" or directly quantify projective compensation. We can detect if it is occurring, but we cannot quantify the exact amount of error that is being projected across each individual parameter; this effect worsens as more parameters are added to our optimization. **As long as there are any errors within the system, there will always be some degree of projective compensation across a global optimization process.**

A least-squares optimization process, as is often used in perception, is inherently designed to minimize the square-sum error across all observations and parameters. Projective compensation is described by the *correlations* between estimated parameters, not the parameters themselves; there aren't any tools within the least-squares optimization itself that can minimize this error any more than is already being done.

Since we can't prevent or minimize projective compensation explicitly, the best we can do is to characterize it by analyzing the *output* of an optimization. In the context of calibration, we're usually doing this as a quality check on the final results, to make sure that the calibration parameters and extrinsic geometry we have computed were estimated as statistically independent entities.

The Giveaway: Parameter Correlations & Covariance

Luckily for us, a least-squares optimization produces both final parameter estimations and a set of variances and covariances for those parameter estimations *and* a set of variances and covariances for those parameter estimations. These covariances act as a smoke-signal: if large, we can bet that the two parameters are highly coupled and exhibiting projective compensation.

Unfortunately, in our example above, we couldn't do this at all! We constrained our problem such that we were not optimizing for the x and y-coordinates of our "known" points. This didn't stop Projective Compensation from affecting our final solution, but it did make it so that we couldn't *observe* when it was occurring since we couldn't directly relate the errors we were witnessing back to a correlation between estimated parameters.

What constitutes a "high correlation" can be somewhat up to interpretation. A good rule of thumb to go by is that if a correlation between two parameters exceeds 0.7, then it is quite large. The 0.5-0.7 range is large, but not egregious, and anything less than 0.5 is a manageable quantity. This is a very broad stroke however, so it doesn't always apply!

This is one reason why Tangram Vision's calibration system optimizes the target field / object space in addition to intrinsics and extrinsics of the cameras in the system. Without doing so, we wouldn't be able to detect these errors or notify users of them at all, but we'd still have to live with the consequences!

Calibration: Projective Compensation In Practice

When it comes to camera calibration, certain projective compensations are well-studied and understood. We can break down the reasons for projective compensation into two categories:

1. A modeling deficiency, e.g. parameters not being entirely statistically independent.
2. A data deficiency in our input data.

Minimizing these projective compensations can be as simple as collecting images from different perspectives (overcoming modeling deficiency), or by increasing the size of the observed target field / checkerboard / targets themselves (overcoming data deficiency).

We describe each of these two scenarios in a bit more detail.

Model Deficiencies

Camera calibration is necessarily driven by model selection. No amount of data will correct your model if it is introducing parameter correlations due to the very nature of the math.

For instance, some camera model parameters are not strictly statistically independent; this means that changes in value for one can affect the other. In some scenarios, this is necessary and expected: in Brown-Conrady distortion, we expect k_1, k_2, k_3 etc. to be correlated, since they describe the same physical effect modeled as a polynomial series. However, this can be troublesome and unnecessary in other scenarios such as e.g. f_x and f_y, which are always highly correlated because they represent the same underlying value, f .

This was somewhat touched upon in our previous chapters, where we discussed different modeling approaches to both projection/focal length, as well as lens distortions.

For now, we'll avoid diving too deep into the intuition behind certain modeling practices, as the specifics are particular to certain camera types and configurations. Just know that choosing the right model is one way we avoid coupling our parameters together, and gives us tools to understand the relationships between the parameters and the data we're collecting.

Data Deficiencies

In our Very Simple Problem at the beginning of the article, we demonstrated both how bad input data can damage the solution, as well as how additional observations could improve it (if not perfectly). Our lack of sufficient input data from that system prevented us from observing any pair of parameters independently, i.e. it affected the observability of a quantity being modeled. Even if a calibration problem is modeled sufficiently and correctly, it is still possible for *observability* to suffer because of bad data.

Here is a (non-exhaustive) list of scenarios where projective compensation occurs in camera calibration due to data deficiencies:

- c_x and c_y may be correlated largely with the *X* and *Y* components of the camera's pose, due to a lack of variation of pose perspectives in the data. This results in extrinsic parameters being highly tied to errors in the determination of the intrinsic parameters, and vice-versa.
- c_x may be highly correlated to c_y, for the same reason as above. The way to handle such correlations is to add data to the adjustment with the camera rotated 90° about the *Z* axis, or rotating the target field / checkerboard 90°.
- f may be highly correlated to all of the extrinsic parameters in the standard model, if all the observations lie within a single plane. This can occur if one never moves a calibration board around much at all. In degenerate cases your calibration optimization won't converge, but if it does, it will be riddled with projective compensations everywhere, and the variances of the estimated quantities will be quite large.

There are more cases similar to the above. In such cases, it is usually sufficient to add more data to the calibration process. What data to add specifically relates to the parameter correlations observed, and are specific to particular kinds of projective compensation.

Read more about the perils of projective compensation in the Tangram Vision Platform Documentation!

Conclusion

Projective compensation is an effect that prominently affects many calibration scenarios. From the simple case described in this article, to much more complex cases seen in practice, an understanding of what projective compensation is and how to detect and address it is fundamental for producing stable and reliable calibrations.

Tangram Vision helps you reduce projective compensation where possible, both by providing support for a selection of models specialized for different camera types, as well as providing best-in-class documentation for understanding the right way to collect data for your needs. We're wrapping this all up in the Tangram Vision Platform, with modules like TVCal for calibration, TVMux for streaming, and the Tangram Vision User Hub for cloud access to our tools. And, as always, check out our Open-Source Software for more perception resources.

COORDINATE FRAMES FOR MULTI-SENSOR SYSTEMS - PART I

FOUNDATIONS FOR BUILDING CALIBRATION

Calibration is an essential tool for any perception-based product. But, to do it right, you need to ground yourself in math and physics.

Multi-sensor systems are becoming ever more common as we seek to automate robotics, navigation, or create a "smart" world. Any random assortment of sensors could be sensing any number of quantities, from barometric pressure, to temperature, to even more complex sensors that scan the world around us and produce 3D maps (like LiDAR). These multi-sensor systems often carry a great deal of complexity, to be able to learn about the world to the same degree that we can just by listening, feeling, and seeing our environment.

An important aspect of multi-sensor systems is how we relate these assorted sensing platforms together. If we are reading temperature, how do we use that value to make decisions? If we have multiple cameras, how can we tell if an object moves from the view of one camera to the next? Individual sensors on their own may be insufficient for making decisions, and therefore need to be linked together. There are a handful of ways we can correlate data, but one of the most powerful ways is to do so spatially. In this post, we're going to explore some of the language and tools we use for doing so.

Location, location, location

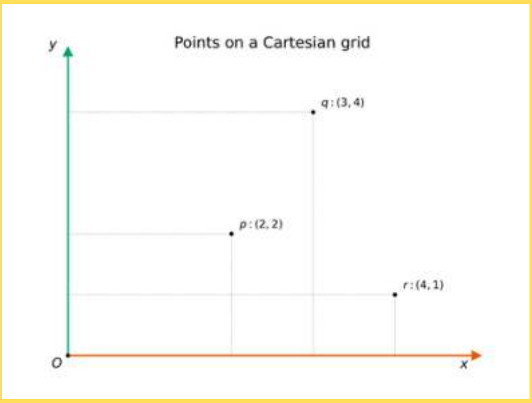
It's all about location. No, really. All sensors have some spatial property, and it is by relating sensors together in this way that we can produce useful results! Knowing that a sensor measured 30° C isn't particularly useful on its own. Knowing that a sensor in your thermostat measured 30° C is a much more useful distinction. The same holds true for more sophisticated sensors such as cameras, LiDAR, accelerometers, etc. These "advanced" sensors are even measuring spatial quantities, which are the backbone of modern robotics and automation.

COORDINATE FRAMES FOR MULTI-SENSOR SYSTEMS

Many of the useful aspects of multi-sensor systems are derived by a spatial understanding of the world around us. Location helps us decide how related any two quantities might be; rather, it is in relating things spatially that our sensors can derive the context of the world around us. Many of the problems in integrating new sensors into a robotics or automation system are therefore coordinate system problems. To understand this, we first need to talk about coordinate frames and then discuss how to relate any two coordinate frames.

Sensors and Coordinate Frames

A coordinate frame or coordinate system is a way for us to label the positions of some points using a set of coordinates, relative to the system's origin. A common type of coordinate frame is a Cartesian coordinate frame, which labels positions along a set of perpendicular axes. Consider the following Cartesian grid, defining a coordinate frame:



An example of a Cartesian grid with three points (*p*, *q*, *r*) plotted within. This grid represents a coordinate frame or coordinate system with an origin *O* and two perpendicular axes *x* and *y*.

For a Cartesian frame, we denote the position of our points as a tuple with the offset from the origin *O*

along each of the axes. In the above example, our point *p* has a coordinate of (2, 2), while q has a coordinate of (3, 4). For any point in the system, you can describe its coordinate as (x_p, y_p) or a point p, where x_p is the offset from the origin O along the direction of the x-axis and y_p is the offset from the origin O along the direction of the y-axis.

Other types of coordinate frames exist as well, including polar coordinate systems and spherical coordinate systems. Map projections such as Mercator or Gnomonic maps are also a type of projected coordinate system, that exist as a handy way to plot and interpret the same data in different ways. For now however, we will focus on Cartesian coordinate frames, as they tend to be the most commonly used coordinate frame to represent and interpret spatial data.

From the example above, we can pick out a few salient components of our Cartesian frame:

- *O*: The origin of our coordinate frame. This tells us to what point every point is relative. Every point is relative to the origin, and the origin can be defined as a point that has zero offset relative to itself.
- *x*- and *y*-axes: Every coordinate frame will have a number of axes used to describe the various dimensions of the coordinate system. For now, we're sticking with 2-dimensional (2D) data, so we'll stick to labeling our axes as the *x*- and *y*-axes. These could actually be called anything, like the *a*- and *b*-axes, but the typical convention for Cartesian coordinate frames is to name them *x* and *y*.
- Axes order: oftentimes you'll hear about left-handed vs. right-handed coordinate systems. There are ways to distinguish the difference, but we're choosing to gloss over that for now in this primer.

Typically, a Cartesian frame is represented as a "right-handed" coordinate frame. The distinction isn't super important, and the idea of left vs. right-handed frames can be extremely confusing at first. More often than not, you won't see a left-handed Cartesian coordinate system that uses *x* / *y* / *z* terminology. Moreover, right-handed frames are more-or-less the standard for conventional robotics sensing. All you need to know is that for what we're doing here we will be representing all our math for right-handed, Cartesian frames.

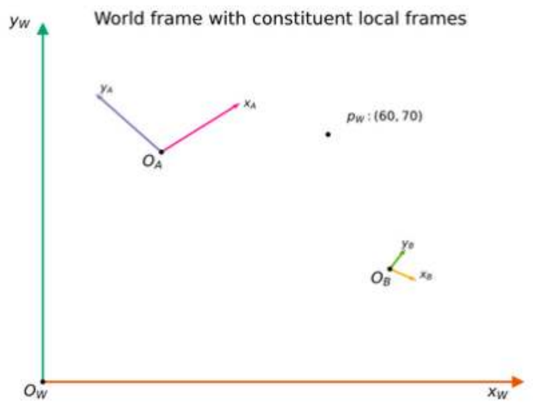
In order to spatially relate our sensors together, we need to give each sensor its own local coordinate frame. For a camera, this might be where the camera lens is centered over the image plane, or what we call the principal point. For an accelerometer, this may be the centre of where accelerations are read. Regardless of where this coordinate frame is defined though, we need to have a "local" frame for each of these sensors so that we can relate these together.

The "World Frame"

When relating multiple coordinate frames together, it is often helpful to visualize a "world" frame. A world frame is a coordinate frame that describes the "world," which is a space that contains all other coordinate frames we care about. The world frame is typically most useful in the context of spatial data, where our "world" may be the real space we inhabit. Some common examples of "world" frames that get used in practice:

- Your automated vehicle platform might reference all cameras (local systems) to an on-board inertial measurement unit (IMU). The IMU may represent the "world" frame for that vehicle.
- A mobile-mapping system may incorporate cameras and LiDAR to collect street-view imagery. The LiDAR system, which directly measures points, may be treated as your world frame.

In the world frame, we can co-locate and relate points from different local systems together. See the below figure, which shows how a point p in the world frame can be related between multiple other frames, e.g. A and B. While we may only know the coordinates for a point in one frame, we eventually want to be able to express the coordinates of that point in other coordinate frames as well.



A world frame, denoted by the orange and teal axes with origin O_W. This world frame contains two "local" coordinate frames A and B, with origins O_A and O_B A common point p is also plotted. This point exists in all three coordinate frames, but is only listed with a coordinate for p_W, which is the point's coordinates in the world frame.

The world frame as shown in the figure above is useful as it demonstrates a coordinate frame in which we can reference the coordinates of O_A and O_B relative to the world frame. All coordinate systems are relative to some position; however, if O_A and O_B are only relative to themselves, we have no framework with which to relate them. So instead, we relate them within our world frame, which helps us visualize the differences between our coordinate systems A and B.

Relating Two Frames Together

In the previous figure, we showed off two local frames A & B inside our world frame W. An important aspect in discussing coordinate frames is establishing a consistent notation in how we refer to coordinate frames and their relationships. Fortunately, mathematics comes to the rescue here as it provides some tools to help us derive a concise way of relating coordinate frames together.

Let's suppose we still have two coordinate frames we care about, namely coordinate frame A and coordinate frame B. We have a coordinate in frame A, p_A, that we want to know the location of in frame B (i.e. we are searching for p_B). We know that this point can exist in both frames, and therefore there is a functional *transformation* to convert to p_B from p_A . In mathematics, we express relations as a function:

$$p_B = f(p_A)$$

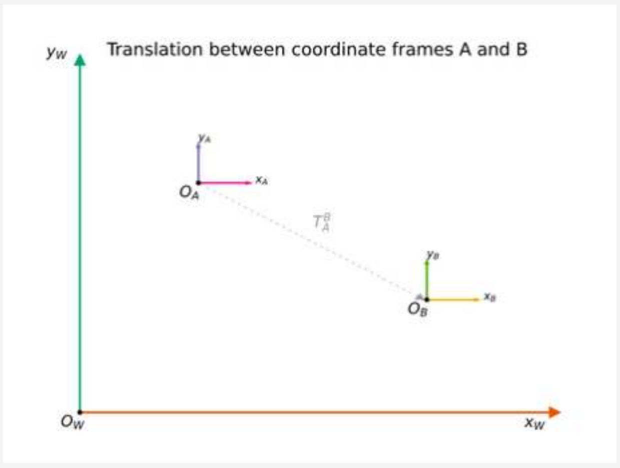
We don't know what this function f is yet, but we will define it soon! It's enough to know that we are getting a point in coordinate frame B from coordinate frame A. The function f is what we typically call a transform. However, saying "the coordinate frame B from coordinate frame A transform" is a bit wordy. We tend to lean towards referring to this transform as the B from A transform, or B←A transform for brevity.

Notice the direction here is not A to B, but rather B from A. While this is a bit of a semantic distinction, the latter is preferred here because of how it is later expressed mathematically. We will eventually define a transformation matrix, Γ^B_A , that describes the transform f above. The superscript and subscript are conventionally written this way, denoting the B←A relationship. Keeping many coordinate frames consistent in your head can be difficult enough, so consistency in our notation and language will help us keep organized.

Now that we have a notation and some language to describe the relationship between two coordinate frames. Fortunately, there's only 3 categories of transformations that we need to care about: translations, rotations, and scale!

Translation

Translations are the easiest type of coordinate transform to understand. In fact, we've already taken translation for granted when defining points in a Cartesian plane as offsets from the origin. Given two coordinate systems A and B, a translation between the two might look like this:



A world frame that shows two local frames, A and B, which are related to each other by a translation. This translation, T^B_A, is just a set of linear offsets from one coordinate frame to the other.

Mathematically, we might represent this as:

$$p_i = \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

$$T_A^B = \begin{bmatrix} T_x \\ T_y \end{bmatrix}_A^B$$

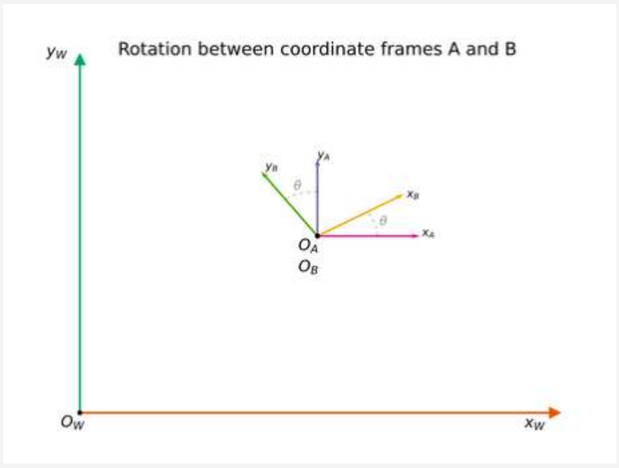
$$p_B = p_A + T_A^B$$

$$\begin{bmatrix} x_B \\ y_B \end{bmatrix} = \begin{bmatrix} x_A \\ y_A \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \end{bmatrix}_A^B$$

Pretty simple addition, which makes this type of transformation easy!

Rotation

Rotations are probably the most complex type of transform to deal with. These transforms are not fixed offsets like translations, but rather vary based on your distance from the origin of your coordinate frame. Given two coordinate frames A and B, a rotation might look like so:



A world frame that shows two local frames, A and B, which are related to each other by a rotation. This rotation, R^B_A, is a linear transformation that depends on the angle of rotation, \theta

In the 2D scenario like shown above, we only have a single plane (the XY-plane), so we only need to worry about a single rotation. Mathematically, we would represent this as:

$$R_A^B = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}_A^B$$

This rotation matrix assumes that positive rotations are counter-clockwise. This is what is often referred to as the right-hand rule.

To get a point p_B from p_A we then do:

$$p_B = R_A^B \cdot p_A$$

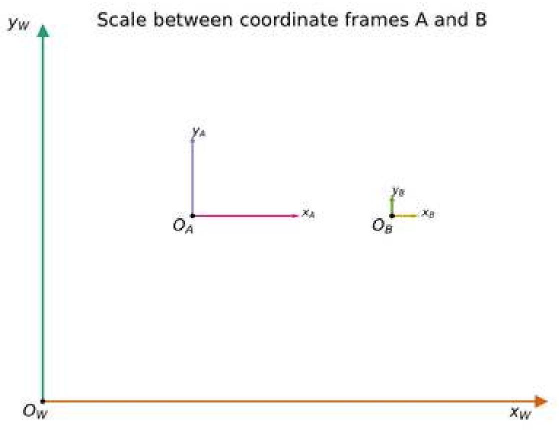
$$\begin{bmatrix} x_B \\ y_B \end{bmatrix} = \begin{bmatrix} R_{xx} & R_{xy} \\ R_{yx} & R_{yy} \end{bmatrix}_A^B \cdot \begin{bmatrix} x_A \\ y_A \end{bmatrix}$$

$$\begin{bmatrix} x_B \\ y_B \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}_A^B \cdot \begin{bmatrix} x_A \\ y_A \end{bmatrix}$$

The matrix multiplication is a bit more involved this time, but remains quite straightforward. Fortunately this is the most difficult transformation to formulate, so we're almost there!

Scale

The final type of transformation we need to consider is scale. Consider the following diagram, similar to the ones we've shown thus far:



A world frame that shows two local frames, A and B, which are related to each other by both a translation and scale. The translation exists to compare A and B side-by-side, but the difference in axes lengths between the A and B frames is a visual trick used to demonstrate what a scale factor might look like, assuming that two coordinate frames are compared in reference to a world frame.

The two frames are again translated, but this is not important for what we're looking at here. Notice that the axes of A are a different length than the axes of B. This is a visual trick to demonstrate what scale transformations do between two coordinate frames. An example of a real-world scale issue might be a unit conversion. Namely, B might be in units of meters, while A is in units of millimeters. This scale difference will change the final results of any point p_B relative to p_A , by a multiplicative factor. If we have an isometric scale, we might represent this mathematically as:

$$s_A^B$$

$$p_B = s_A^B \cdot p_A$$

Now, in this way, we are using a scalar value to represent an isometric scale across both x and y axes. This is not always the case, as sometimes our scale is not isometric. In robotics, we typically treat most of our sensors as having an isometric scale, but it is worth showing the mathematics for how one might generalize this if the scale in x (s_x) is different from the scale in y (s_y):

$$S_A^B = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}_A^B$$

$$p_B = S_A^B \cdot p_A$$

$$\begin{bmatrix} x_B \\ y_B \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}_A^B \cdot \begin{bmatrix} x_A \\ y_A \end{bmatrix}$$

By utilizing this more general matrix equation over the scalar form above, it is easy to abstract between isometric scales (where $s_x = s_y$), and affine transforms. Fortunately it's often very easy to get away with assuming that our scale is isometric.

Putting It All Together

Now that we know the three types of transforms, how do we put it all together? Any two coordinate frames A and B could have any number of translations, rotations, and scale factors between them. A transform in the general sense incorporates all three of these operations, and so we need a more formal way to represent them. Using our previous notation, we can formulate it as follows:

$$p_B = f(p_A)$$

$$p_B = S_A^B \cdot R_A^B \cdot (p_A + T_A^B)$$

$$\begin{bmatrix} x_B \\ y_B \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}_A^B \cdot \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}_A^B \cdot \begin{bmatrix} x_A \\ y_A \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \end{bmatrix}_A^B$$

Keep in mind the order here:

1. Rotation
2. Scale
3. Translation

With all the multiplications going on, this can get very confusing very quickly! Moreover, remembering the order every time can be pretty difficult. To make this more consistent, mathematicians and engineers often try to represent it as a single matrix multiplication. This way, the order is never confusing. We call this single matrix Γ_A^B. In plain English, we typically call this the B from A transformation matrix, or just the B⟵A transform. Unfortunately, however, you'll notice that not every operation in our above equation is a matrix multiplication, so the following doesn't work!

$$p_B = \Gamma_A^B \cdot p_A$$

This would be fine for rotation and scale, but doesn't allow us to do anything about translations. Fortunately, we can work around this somewhat by leveraging a small trick of mathematics: increasing the dimensionality of our problem. If we change some of our definitions around, we can create a nuisance or *dummy* dimension that allows us to formulate Γ_A^B as:

$$p_i = \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

$$\Gamma_A^B = \begin{bmatrix} s_x R_{xx} & s_x R_{xy} & T_x \\ s_y R_{yx} & s_y R_{yy} & T_y \\ 0 & 0 & 1 \end{bmatrix}_A^B$$

$$p_B = \Gamma_A^B \cdot p_A$$

$$\begin{bmatrix} x_B \\ y_B \\ 1 \end{bmatrix} = \begin{bmatrix} s_x cos(\theta) & -s_x sin(\theta) & T_x \\ s_y sin(\theta) & s_y cos(\theta) & T_y \\ 0 & 0 & 1 \end{bmatrix}_A^B \cdot \begin{bmatrix} x_A \\ y_A \\ 1 \end{bmatrix}$$

Notice that our last dimension on each point remains equal to 1 on both sides of the transformation (this is our nuisance dimension)! Additionally, the last row of Γ_A^B is always zeros, except for the value in the bottom right corner of the matrix, which is also a 1!

If you want to learn more about the trick we're applying above, search for homogeneous coordinates or projective coordinates!

Try it for yourself with these Python functions! Find this and the code used to generate our above figures in our Tangram Visions Blog repository.

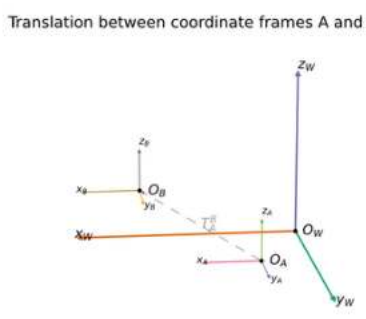
What was it all for?

Knowing how to express relationships between coordinate frames both in plain English (I want the B⟵A or B from A transformation) and in mathematics (I want Γ_A^B helps bridge the gap for how we relate sensors to each other. In a more concrete example, suppose we have two cameras: depth and color. We might want depth⟵color, so that we can fuse semantic information from the color camera with spatial information from the depth camera. Eventually, we want to then relate that information back to real world coordinates (e.g. I want world←depth).

Coordinate frames and coordinate systems are a key component to integrating multi-sensor frameworks into robotics and automation projects. Location is of the utmost importance, even more when we consider that many robotics sensors are spatial in nature. Becoming an expert in coordinate systems is a path towards a stable, fully-integrated sensor suite on-board your automated platform of choice.

While these can be fascinating challenges to solve while creating a multi-sensor-equipped system like a robot, they can also become unpredictably time consuming, which can delay product launches and feature updates. The Tangram Vision Platform includes tools and systems to make this kind of work more streamlined and predictable.

COORDINATE SYSTEMS, AND HOW TO RELATE MULTIPLE COORDINATE FRAMES TOGETHER. (PART 2)



Translation between two 3D coordinate frames, A and B.

Rotate, Scale, Translate: Coordinate frames for multi-sensor systems (Part 2)

In the previous chapter, we covered how rotations, scales, and translations work for 2D coordinate frames. The most important takeaway was that we were able to both think conceptually about transformations, as well as express them in plain English (I want the depth from color transform) and mathematically (I want Γ^{depth}_{color} , or depth ← color).

While 2D coordinate frames are common in mathematics, we interact with our world in three dimensions (3D). In particular, many of the sensors that Tangram Vision deals with every day have to be related to each other in 3D space. This means that most of the coordinate systems we are interested in tend to be expressed in three dimensions as well. Relating two 3D coordinate systems together must be done regardless of how complex your sensor system is. That includes anything from orienting two cameras into the same frame, to combining LiDAR with other sensors. This means that the 2D transforms we derived last time won't be enough.

In this article, we aim to extend the previous equations we formulated for 2D transforms and derive equivalent 3D transforms. Let's get started!

You can follow along here, or at the full codebase at the Tangram Visions Blog repository. Let's get started!

Extending Our Transformations to 3D

Recall that we try to express the entire transformation process as a function applied over a point. We started with the following equation:

$$p_B = f(p_A)$$

Fortunately, this model still works! In 3D, our points have three Cartesian axes: X, Y, and Z. Ultimately, we are looking for one of the following forms:

$$p_B = \begin{bmatrix} x_B \\ y_B \\ z_B \end{bmatrix} = S_A^B \cdot R_A^B \cdot \begin{bmatrix} x_A \\ y_A \\ z_A \end{bmatrix} + T_A^B$$

or, alternatively with projective coordinates:

$$p_B = \begin{bmatrix} x_B \\ y_B \\ z_B \\ 1 \end{bmatrix} = \Gamma_A^B \cdot \begin{bmatrix} x_A \\ y_A \\ z_A \\ 1 \end{bmatrix}$$

As we did last time, let's look at each of these coordinate transformations independently, and then see how they combine together!

Translation

Just like last time, translations are the easiest transformations to understand. We add an additional translation dimension just as we did with our definition of points above.

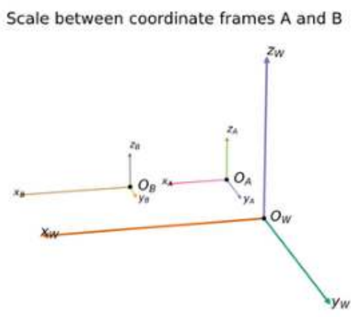
$$T_A^B = \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix}_A^B$$

$$p_B = p_A + T_A^B$$

$$\begin{bmatrix} x_B \\ y_B \\ z_B \end{bmatrix} = \begin{bmatrix} x_A \\ y_A \\ z_A \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix}_A^B$$

Again, it comes out to be simple addition, which keeps things easy.

Scale



Scaling between two 3D coordinate frames, A and B. Notice the different lengths of the axes between A and B.

Scaling to an additional dimension changes the final equation similarly to how it did with translation. We add an extra element to the final matrix, producing:

$$S_A^B = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}_A^B$$

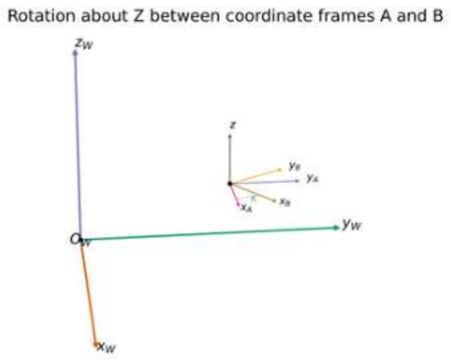
$$p_B = S_A^B \cdot p_A$$

$$\begin{bmatrix} x_B \\ y_B \\ z_B \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}_A^B \cdot \begin{bmatrix} x_A \\ y_A \\ z_A \end{bmatrix}$$

Just like last time, we may have an isometric scale $(s_x = s_y = s_z)$, or we might have an affine transform of some form ($(s_x \neq s_y \neq s_z)$. Fortunately, our scale matrix remains simple to write and apply!

Rotation

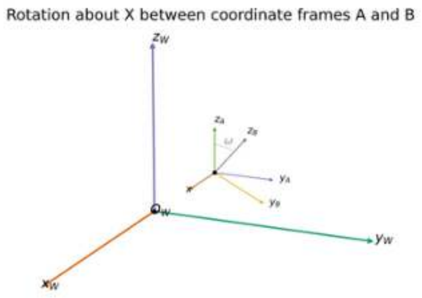
Rotation is again going to be the most complex out of our transformations. First, let's extend our rotation matrix from last time into three dimensions. Recall that we were rotating the X and Y dimensions (i.e. we were rotating about the Z axis):



Rotation about the Z axis between two 3D coordinate frames A and B, centered on the same origin point. See the rotation angle κ between the respective axes of the two coordinate frames.

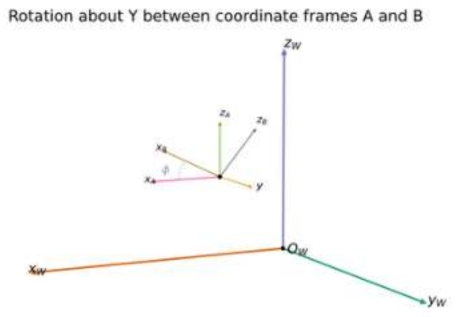
$$R_z(\kappa)_A^B = \begin{bmatrix} cos(\kappa) & -sin(\kappa) & 0 \\ sin(\kappa) & cos(\kappa) & 0 \\ 0 & 0 & 1 \end{bmatrix}_A^B$$

Notice we called this the $R_z(\kappa)$ matrix above, as it encodes rotations of angle κ about the Z axis. The challenge we find ourselves with is that this matrix can only rotate about the Z axis! It isn't possible to perform any rotations of points in the XZ or YZ planes with this matrix. For these, we would need to rotate about the Y and X axes, respectively. For this we define two new rotation matrices, $R_x(\omega)$ and $R_y(\phi)$.



Rotation about the X axis between two 3D coordinate frames A and B, centered on the same origin point. See the rotation angle ω between the respective axes of the two coordinate frames.

$$R_x(\omega)_A^B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\omega) & -sin(\omega) \\ 0 & sin(\omega) & cos(\omega) \end{bmatrix}_A^B$$



Rotation about the Y axis between two 3D coordinate frames A and B, centered on the same origin point. See the rotation angle ϕ between the respective axes of the two coordinate frames.

$$R_y(\phi)_A^B = \begin{bmatrix} cos(\phi) & 0 & sin(\phi) \\ 0 & 1 & 0 \\ -sin(\phi) & 0 & cos(\phi) \end{bmatrix}_A^B$$

Each of these rotation matrices R_x, R_y, and R_z rotate about their respective axes by some angle ω, ϕ, or κ. Colloquially, these ω, ϕ, and κ are often referred to as *roll*, *pitch*, and *yaw* angles. We often refer to these as a set of *Euler Angles* when referring to the entire set. With all these matrices, we're able to encode any arbitrary rotation between two coordinate frames. The question is: how do we combine them?

Unfortunately, the answer is not so clear. Different systems may use different conventions to combine these rotation matrices. While the end result may be the same, the underlying angles and matrix components will not be. Typically, picking an order to apply these matrices is called picking a parameterization. Which parameterization you pick is likely to be driven by your application.

One aspect that can drive which parameterization you pick is a concept known as gimbal lock. Gimbal lock is a singularity in our parameterization that makes it impossible to extract our original Euler Angles from a combined rotation matrix. That may not make a lot of sense, so here's an example:

Suppose we decided to choose the X Y Z parameterization, this would look like:

$$R_{xyz}(\omega,\phi,\kappa)^B_A = R_x(\omega)^B_A \cdot R_y(\phi)^B_A \cdot R_z(\kappa)^B_A =$$
$$\begin{bmatrix} cos(\phi)cos(\kappa) & -cos(\phi)sin(\kappa) & sin(\phi) \\ cos(\omega)sin(\kappa)+cos(\kappa)sin(\omega)sin(\phi) & cos(\omega)cos(\kappa)-sin(\omega)sin(\phi)sin(\kappa) & -cos(\phi)sin(\omega) \\ sin(\omega)sin(\kappa)-cos(\omega)cos(\kappa)sin(\phi) & cos(\kappa)sin(\omega)+cos(\omega)sin(\phi)sin(\kappa) & cos(\omega)cos(\phi) \end{bmatrix}^B_A$$

This is quite the equation! To make things easier, we might instead write the rotation matrix with some placeholders so that we don't have to write every equation every time (and risk getting it wrong).

$$R_{xyz}(\omega,\phi,\kappa)^B_A = \begin{bmatrix} r_{xx} & r_{xy} & r_{xz} \\ r_{yx} & r_{yy} & r_{yz} \\ r_{zx} & r_{zy} & r_{zz} \end{bmatrix}^B_A$$

For starters, lets extract the ϕ angle from this matrix above:

$$r_{xz} = sin(\phi)$$

$$sin^{-1}(r_{xz}) = \phi$$

This is fine, except when ϕ is $\frac{\pi}{2}$ or $-\frac{\pi}{2}$ (90° or 270°). If we expand the matrix to actual values assuming that $\phi = \frac{\pi}{2}$, we get:

$$R_{xyz}(\omega,\phi,\kappa)^B_A = \begin{bmatrix} 0 & 0 & 1 \\ sin(\omega+\kappa) & cos(\omega+\kappa) & 0 \\ -cos(\omega+\kappa) & sin(\omega+\kappa) & 0 \end{bmatrix}$$

This means that regardless of whether we change ω or κ, we have the same effect. Likewise, it means that there is no way to individually extract these two values as we cannot differentiate between the two of them! This is what is meant when someone states that there is a gimbal lock.

Gimbal lock motivates how one might choose a parameterization, to try and avoid it. There are 12 in all, as described in this paper from NASA. The most common choices of parameterization you might see are the XYZ and ZYX parameterizations, but others such as ZYZ or XYX are also quite common. When choosing a parameterization, remember that the resulting rotation matrix is just a combination of the individual X, Y, and Z rotation matrices that were defined above!

There are other parameterizations for rotations (such as unit quaternions or the geometric algebra) that don't have this problem with gimbal lock at all. We haven't covered these, but they provide an alternative if you're finding this 3D representation of transformations problematic.

Putting it all together

Like before, let's put this all together! Regardless of how we parameterize our rotations, we get the following relationship:

$$p_B = f(p_A)$$

$$p_B = S_A^B \cdot R_A^B \cdot p_A + T_A^B$$

$$\begin{bmatrix} x_B \\ y_B \\ z_B \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \cdot \begin{bmatrix} r_{xx} & r_{xy} & r_{xz} \\ r_{yx} & r_{yy} & r_{yz} \\ r_{zx} & r_{zy} & r_{zz} \end{bmatrix} \cdot \begin{bmatrix} x_A \\ y_A \\ z_A \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix}$$

Just like last time, the order is:

1. Rotation
2. Scale
3. Translation

And just like last time, we can use projective or homogeneous coordinates to convert this into a single Γ_A^B matrix:

$$p_i = \begin{bmatrix} x_i & y_i & z_i & 1 \end{bmatrix}^T$$

$$\Gamma_A^B = \begin{bmatrix} s_x r_{xx} & s_x r_{xy} & s_x r_{xz} & T_x \\ s_y r_{yx} & s_y r_{yy} & s_y r_{yz} & T_y \\ s_z r_{zx} & s_z r_{zy} & s_z r_{zz} & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}_A^B$$

$$p_B = \Gamma_A^B \cdot p_A$$

$$\begin{bmatrix} x_B \\ y_B \\ z_B \\ 1 \end{bmatrix} = \begin{bmatrix} s_x r_{xx} & s_x r_{xy} & s_x r_{xz} & T_x \\ s_y r_{yx} & s_y r_{yy} & s_y r_{yz} & T_y \\ s_z r_{zx} & s_z r_{zy} & s_z r_{zz} & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}_A^B \cdot \begin{bmatrix} x_A \\ y_A \\ z_A \\ 1 \end{bmatrix}$$

Just as in the 2D case, the final dimension on our points remains 1, even after multiplying by Γ_A^B !

Moving from 2D to 3D

In this article we demonstrated how our language and mathematical models for understanding 2D coordinate frames can easily and readily be extended into three dimensions with minimal effort. Most importantly, we managed to preserve our ability to talk about coordinate transformations at a high level in English (I want the B from A transform) as well as mathematically (I want Γ_A^B).

Relating 3D coordinate frames is a cornerstone component of any multi-sensor system. Whether we are relating multiple cameras, or stitching disparate sensing technologies together, it is all about location and where these sensors are relative to one another. Unlike the 2D case, we discovered that 3D rotations need to consider the problem of parameterization to avoid a gimbal lock. This is something that all sensing systems need to consider if they want to use the matrix representation that we provide here. In fact, Tangram Vision uses these same mathematics when reasoning about device calibration in systems with much more complexity.

The world of coordinates and mathematics can be both fascinating and complex when working in multi-sensor environments. However, if this all seems like complexity that you would rather trust with experts, The Tangram Vision Platform includes tools and abstractions that simplify optimizing multi-sensor systems for rapid integration and consistent performance.

ONE-TO-MANY SENSOR TROUBLE, PART 1

Or, how to Kalman filter your way out

In the world of automated vision, there's only so much one can do with a single sensor. Optimize for one thing, and lose the other; and even with one-size-fits-all attempts, it's hard to paint a full sensory picture of the world at 30fps.

We use sensor combinations to overcome this restriction. This intuitively seems like the right play; more sensors mean more information. The jump from one camera to two, for instance, unlocks binocular vision, or the ability to see behind as well as in front. Better yet, use three cameras to do both at once. Add in a LiDAR unit, and see farther. Add in active depth, and see with more fidelity. Tying together multiple data streams is so valuable this act of **Sensor Fusion** is a whole discipline in itself.

Yet this boon in information often makes vision-enabled systems harder to build, not easier. Binocular vision relies on stable intrinsic and extrinsic camera properties, which cameras don't have. Depth sensors lose accuracy with distance. A sensor can fail entirely, like LiDAR on a foggy day.

This means that effective sensor fusion involves constructing vision architecture in a way that *minimizes uncertainty in uncertain conditions*. Sensors aren't perfect, and data can be noisy. It's the job of the engineer to sort this out and derive assurances about what is actually true. This challenge is what makes sensor fusion so difficult: it takes competency in information theory, geometry, optimization, fault tolerance, and a whole mess of other things to get right.

So how do we start?

Guessing

...just kidding. Though you would be surprised how many times an educated guess gets thrown in! No, we're talking...

Kalman filters

So let's review our predicament:

- We have a rough idea of our current **state** (e.g. the position of our robot), and we have a **model** of how that state changes through time.
- We have **multiple sensor modalities**, each with their own data streams.
- All of these sensors give **noisy and uncertain data**.

Nonetheless, this is all that we have to work with. This seems troubling; we can't be certain about anything!

Instead, what we can do is *minimize our uncertainty*. Through the beauty of mathematics, we can combine all of this knowledge and actually come out with a more certain idea of our state through time than if we used any one sensor or model.

This is the magic of Kalman filters.

Warning: Math.

Phase 1: Prediction

Model Behavior

Let's pretend that we're driving an RC car in a completely flat, very physics-friendly line.
There are two things that we can easily track about our car's state: its position p_t and velocity v_t.

We can speed up our robot by punching the throttle, something we do frequently. We do this by exerting a force f on the RC car's mass m, resulting in an acceleration a (see Newton's Second LawS of Motion).

With just this information, we can derive a model for how our car will act over a time period Δt using some classical physics:

$$p_{t+1} = p_t + (v_t \Delta t) + \frac{f \Delta t^2}{2m} \qquad (1.1)$$

$$v_{t+1} = v_t + \frac{f \Delta t}{m} \qquad (1.2)$$

We can simplify this for ourselves using some convenient matrix notation. Let's put the values we can track, position p_t and velocity v_t, into a **state vector**:

$$\mathbf{x}_t = \begin{bmatrix} p_t \\ v_t \end{bmatrix} \qquad (1.3)$$

...and let's put out applied forces into a **control vector** that represents all the outside influences affecting our state:

$$\mathbf{u}_t = \left[\frac{f}{m}\right] = \frac{f}{m} \qquad (1.4)$$

Now, with a little rearranging, we can organize our motion model for position and velocity into something a bit more compact:

$$\begin{bmatrix} p_{t+1} \\ v_{t+1} \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}}_{F_t} \begin{bmatrix} p_t \\ v_t \end{bmatrix} + \underbrace{\begin{bmatrix} \frac{\Delta t^2}{2} \\ \Delta t \end{bmatrix}}_{B_t} \frac{f}{m} \qquad (1.5)$$

$$\Rightarrow \mathbf{x}_{t+1} = F_t \mathbf{x}_t + B_t \mathbf{u}_t \qquad (1.6)$$

By rolling up these terms, we get some handy notation that we can use later:

- F_t is called our **prediction matrix**. It models what our system would do over Δt , given its current state.
- B_t is called our **control matrix**. This relates the forces in our control vector $\mathbf{u}_t$ to the state prediction over Δt .

Uncertainty through PDFs

However, we're not exactly sure whether or not our state values are true to life; there's uncertainty! Let's make some assumptions about what this uncertainty might look like in our system:

- Any error we might get in an observation is inherently random; that is, there isn't a bias towards one result.
- Errors are independent of one another.

These two assumptions mean that our uncertainty follows the Central Limit Theorem! We can therefore assume that our error follows a Normal Distribution, aka a Gaussian curve.

We will use our understanding of Gaussian curves later to great effect, so take note!

We're going to give this uncertainty model a special name: a **probability density function (PDF)**. This represents how probable it is that certain states are the *true* state. Peaks in our function correspond to the states that have the highest probability of occurrence.

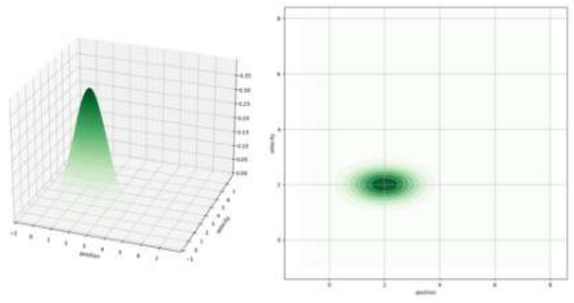


Fig. 1. Our first PDF

Our state vector $\mathbf{x}_t$ represents the mean μ of this PDF. To derive the rest of the function, we can model our state uncertainty using a covariance matrix $\mathbf{P}_t$:

$$\mathbf{P}_t = \begin{bmatrix} \Sigma_{pp} & \Sigma_{pv} \\ \Sigma_{vp} & \Sigma_{vv} \end{bmatrix} \tag{2.1}$$

There are some interesting properties here in $\mathbf{P}_t$. The diagonal elements $(\Sigma_{pp}, \Sigma_{vv})$ represent how much these variables deviate from their own mean. We call this **variance**.

The off-diagonal elements of $\mathbf{P}_t$ express **covariance** between state elements. If Σ_{pv} is zero, for instance, then we know that an error in velocity won't influence an error in position. If it's any other value, we can safely say that one affects the other in some way. PDFs without covariance terms look like Figure 1 above, with major and minor axes aligned with our world axes. PDFs *with* covariance are skewed off-axis depending on how extreme the covariance is:

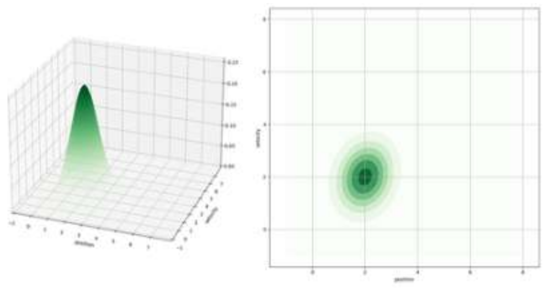


Fig. 2. A PDF with non-zero covariance. Notice the 'tilt' in the major and minor axes of the ellipsis.

Variance, covariance, and the related **correlation** of variables are valuable, as they make our PDF more information-dense.

We know how to predict $\mathbf{x}_{t+1}$, but we also need the predicted covariance $\mathbf{P}_{t+1}$ if we're going to describe our state fully. We can derive it from $\mathbf{x}_{t+1}$ using some (drastically simplified) linear algebra:

$$\mathbf{P}_{t+1} = cov(F_t\mathbf{x}_t) + cov(B_t\mathbf{u}_t) = F_t\mathbf{P}_tF_t^T + \cancel{cov(B_t\mathbf{u}_t)} \tag{2.2}$$

Notice that $B_t\mathbf{u}_t$ got tossed out! Control has no uncertainty that we can directly observe, so we can't use the same math that we did on $F_t\mathbf{x}_t$.

However, we can factor in the effects of noisy control inputs another way: by adding a **process noise covariance matrix** Q_t:

$$\mathbf{P}_{t+1} = F_t\mathbf{P}_tF_t^T + Q_t \tag{2.3}$$

Yes, we are literally adding noise.

Prediction step, solved?

We have now derived the full **prediction step**:

$$\mathbf{x}_{t+1} = \underbrace{F_t\mathbf{x}_t}_{state} + \underbrace{B_t\mathbf{u}}_{control} \tag{2.4}$$

$$\mathbf{P}_{t+1} = \underbrace{F_t\mathbf{P}_tF_t^T}_{prediction} + \underbrace{Q_t}_{process} \tag{2.5}$$

Our results are... ok. We got a good guess at our new state out of this process, sure, but we're a lot more uncertain than we used to be!

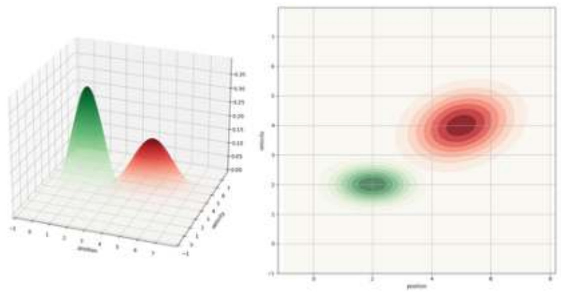


Fig. 3. Starting state PDF in red, predicted state PDF in blue. Notice how the distribution is more spread out. Our state is less certain than before.

There's a good reason for that: everything up to this point has been a sort of "best guess". We have our state, and we have a model of how the world works; all that we're doing is using both to predict what might happen over time. We still need something to **support these predictions** outside of our model.

Something like sensor measurements, for instance.

We’re getting there! So far, this post has covered

- the motivation for using a Kalman filter in the first place
- a toy use case, in this case our physics-friendly RC car
- the Kalman filter prediction step: our best guess at where we’ll be, given our state

We’ll keep it going in the next chapter by bringing in our sensor measurements (finally). We will use these measurements, along with our PDFs, to uncover the true magic of Kalman filters!

Spoiler: it’s not magic. It’s just more math.

ONE-TO-MANY SENSOR TROUBLE, PART 2

Phase 2: Updating Your Prediction

From State to Sensors

While we were busy predicting, the GPS on our RC car was giving us positional data updates. These **sensor measurements** $\mathbf{z}_t$ give us valuable information about our world.

However, $\mathbf{z}_t$ and our state vector $\mathbf{x}_{t+1}$ may not actually correspond; our measurements might be in one space, and our state in another! For instance, what if we're measuring our state in meters, but all of our measurements are in feet? We need some way to remedy this.

Let's handle this situation by converting our state vector into our measurement space using an **observation matrix *H***:

$$\mu_{exp} = H\mathbf{x}_{t+1} \tag{3.1}$$

$$\Sigma_{exp} = H\mathbf{P}_{t+1}H^T \tag{3.2}$$

These equations represent the mean μ_{exp} and covariance Σ_{exp} of our **predicted measurements**. For our RC car, we're going from meters to feet, in both position and velocity. 1 meter is around 3.28 ft, so we would shape ***H*** to reflect this:

$$\mu_{exp} = \begin{bmatrix} 3.28 & 0 \\ 0 & 3.28 \end{bmatrix} \begin{bmatrix} p_{t+1} \\ v_{t+1} \end{bmatrix} \tag{3.3}$$

$$\Sigma_{exp} = \begin{bmatrix} 3.28 & 0 \\ 0 & 3.28 \end{bmatrix} \mathbf{P}_{t+1} \begin{bmatrix} 3.28 & 0 \\ 0 & 3.28 \end{bmatrix} \tag{3.4}$$

...leaving us with predicted measurements that we can now compare to our sensor measurements $\mathbf{z}_t$. Note that ***H*** is entirely dependent on what's in your state and what's being measured, so it can change from problem to problem.

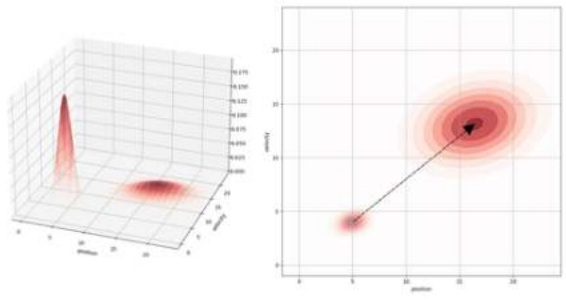


Fig. 1: Moving our state from state space (meters, bottom left PDF) to measurement space (feet, top right PDF).

Our RC car example is a little simplistic, but this ability to translate our **predicted state into predicted measurements** is a big part of what makes Kalman filters so powerful. We can effectively compare our state with any and all sensor measurements, from any sensor. That's powerful stuff!

The behavior of ***H*** is important. Vanilla Kalman filters use a linear ***Ft*** and ***H*** ; that is, there is only one set of equations relating the estimated state to the predicted state (F_t), and predicted state to predicted measurement (***H***). If the system is non-linear, then this assumption doesn't hold. F_t and ***H*** might change every time our state does! This is where innovations like the Extended Kalman Filter (EKF) and the Unscented Kalman Filter come into play. EKFs are the de facto standard in sensor fusion for this reason.

Let's add one more term for good measure: R_t , our sensor measurement covariance. This represents the noise from our measurements. Everything is uncertain, right? It never ends.

The Beauty of PDFs

Since we converted our state space into the measurement space, we now have 2 comparable Gaussian PDFs:

- μ_{exp} and Σ_{exp} , which make up the Gaussian PDF for our predicted measurements
- $\mathbf{z}_t$ and R_t, which make up the PDF for our sensor measurements

The strongest probability for our future state is the overlap between these two PDFs. How do we get this overlap?

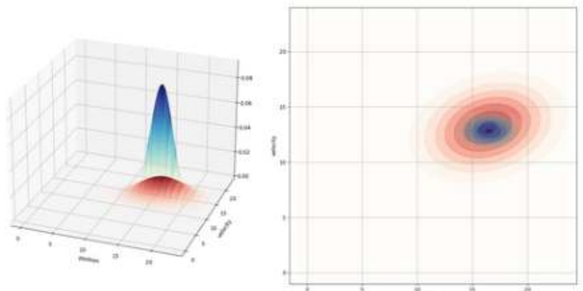


Fig. 2: Our predicted state in measurement space is in red. Our measurements z are in blue. Notice how our measurements z have a much smaller covariance; this is going to come in handy.

We multiply them together! The product of two Gaussian functions is just another Gaussian function. Even better, this common Gaussian has a *smaller covariance* than either the predicted PDF or the sensor PDF, meaning that our state is now much more certain.

ISN'T THAT NEAT.

Update Step, Solved.

We're not out of the woods yet; we still need to derive the math! Suffice to say... it's a lot. The basic gist is that multiplying two Gaussian functions results in its own Gaussian function. Let's do this with two PDFs now, Gaussian functions with means μ_1, μ_2 and variances σ_1^2, σ_2^2 . Multiplying these two PDFs leaves us with our final Gaussian PDF, and thus our final mean and covariance terms:

$$\mu_{final} = \mu_1 + \frac{H\sigma_1^2(\mu_2 - H\mu_1)}{H^2\sigma_1^2 + \sigma_2^2} \tag{3.5}$$

$$\sigma_{final}^2 = \sigma_1^2 - \frac{H\sigma_1^2}{H^2\sigma_1^2 + \sigma_2^2} H\sigma_1^2 \tag{3.6}$$

When we substitute in our derived state and covariance matrices, these equations represent the **update step** of a Kalman filter.

$$\Rightarrow \mathbf{x}_{final} = \mathbf{x}_{t+1} + \frac{H\mathbf{P}_{t+1}(\mathbf{z}_t - H\mathbf{x}_{t+1})}{H^2\mathbf{P}_{t+1} + \mathbf{R}_t} \tag{3.7}$$

$$\Rightarrow \mathbf{P}_{final} = \mathbf{P}_{t+1} - \frac{H\mathbf{P}_{t+1}}{H^2\mathbf{P}_{t+1} + \mathbf{R}_t} H\mathbf{P}_{t+1} \tag{3.8}$$

See reference (Bromiley, P.A.) for an explanation on how to multiply two PDFs and derive the above. It takes a good page to write out; you've been warned.

This is admittedly pretty painful to read as-is. We can simplify this by defining the Kalman Gain K_t :

$$K_t = \frac{H\sigma_1^2}{H^2\sigma_1^2 + \sigma_2^2} \quad (3.9)$$

With K_t in the mix, we find that our equations are much kinder on the eyes:

$$\mathbf{x}_{final} = \mathbf{x}_{t+1} + K_t(\mathbf{z}_t - H\mathbf{x}_{t+1}) \quad (3.10)$$

$$\mathbf{P}_{final} = \mathbf{P}_{t+1} - K_t H \mathbf{P}_{t+1} \quad (3.11)$$

We've done it! Combined, $\mathbf{x}_{final}$ and $\mathbf{P}_{final}$ create our final Gaussian distribution, the one that crunches all of that data to give us our updated state. See how our updated state spikes in probability (the blue spike in Fig. 3). That's the power of Kalman Filters!

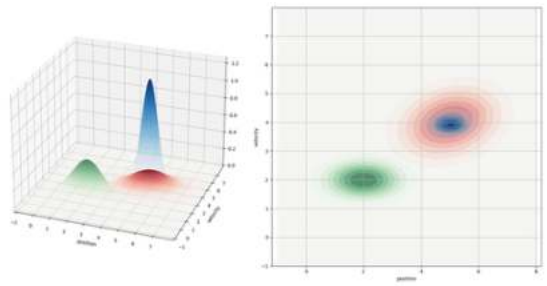


Fig. 3: Original state in green. Predicted state in red. Updated final state in blue. Look at how certain we are now! Such. Wow.

What now?

Well... do it again! The Kalman filter is recursive, meaning that it uses its output as the next input to the cycle. In other words, $\mathbf{x}_{final}$ is your new $\mathbf{x}_t$! The cycle continues in the next prediction state.

Kalman filters are great for all sorts of reasons:

- They extend to anything that can be modeled and measured. Automotives, touchscreens, econometrics, etc etc.
- Your sensor data can come from anywhere. As long as there's a connection between state and measurement, new data can be folded in.
- Kalman filters also allow for the selective use of data. Did your sensor fail? Don't count that sensor for this iteration! Easy.
- They are a good way to model prediction. After completing one pass of the prediction step, just do it again (and again, and again) for an easy temporal model.

Of course, if this is too much and you'd rather do... anything else, Tangram Vision is developing solutions to these very same problems! We're creating tools that help any vision-enabled system operate to its best. If you like this article, you'll love seeing what we're up to.

Code Examples and Graphics

The code used to render these graphs and figures is hosted on Tangram Vision's public repository for the blog. Head down, check it out, and play around with the math yourself! If you can improve on our implementations, even better; we might put it in here. And be sure to tweet at us with your improvements.

Further Reading

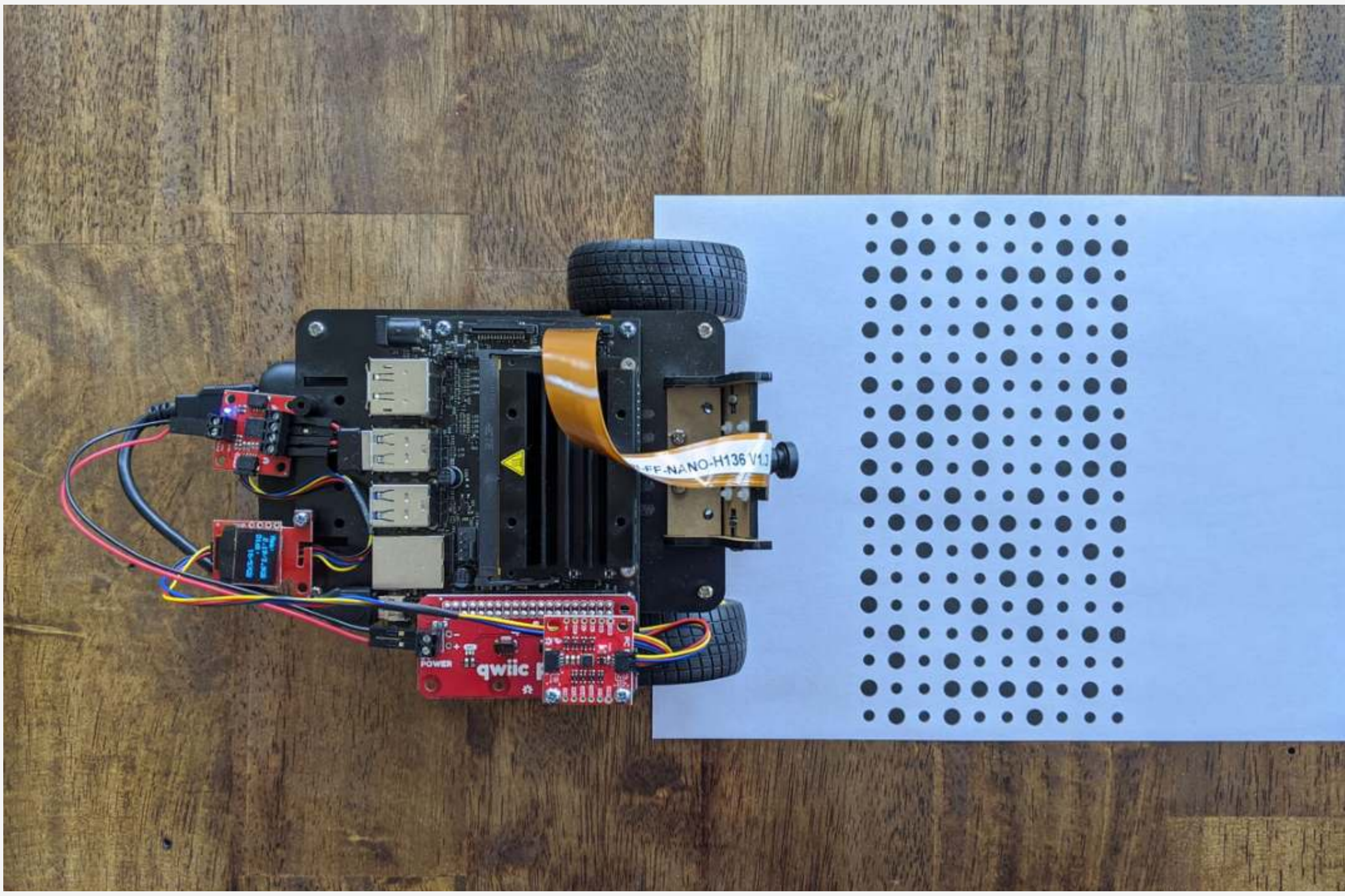
If this article didn't help, here are some other sources that dive into Kalman Filters in similar ways:

1. Faragher, R. (2012, September). Understanding the Basis of the Kalman Filter. https://drive.google.com/file/d/1nVtDUrfcBN9zwKlGuAclK-F8Gnf2M_to/view
2. Bzarg. How a Kalman filter works, in pictures. https://www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/
3. Teammco, R. Kalman Filter Simulation. https://www.cs.utexas.edu/~teammco/misc/kalman_filter/
4. Bromiley, P.A. Products and Convolutions of Gaussian Probability Density Functions. https://bit.ly/3nGDewe [PDF]

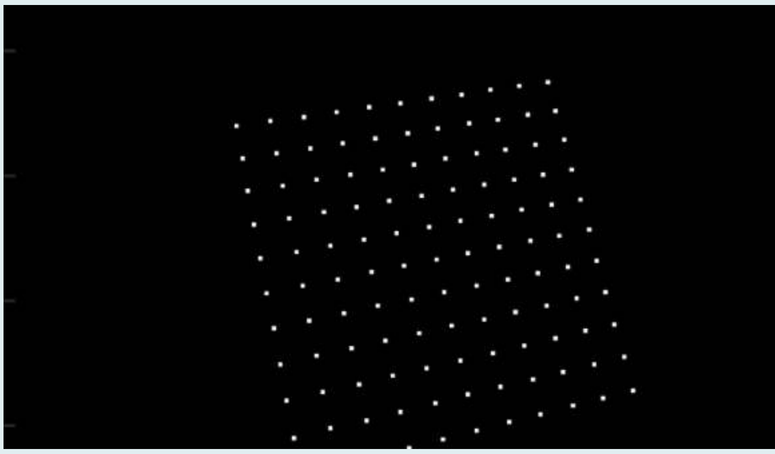
SECTION II: CALIBRATION IMPLEMENTATION

47 / Building Calibration From Scratch (Using Rust)

61 / Reverse Engineering Fiducial Markers For Perception



BUILDING CALIBRATION FROM SCRATCH*



What's involved in creating a complete calibration pipeline? In this in-depth guide, we'll cover the theory behind calibration, the mathematics and principles, and then we'll show the results from a self-built calibration module.

Camera calibration is generally understood to be the process of estimating the characteristics of a camera that affect the formation of images of real 3D scenes. Principle among these is the effective focal length of the lens which gives images a wide-angle or telescopic character for a given camera position and scene-being-image.

There are a wide range of models used to describe a camera's image formation behavior. Some are relatively simple with mostly-linear equations of few components and are only adequate to describe simple camera systems. Others are highly non-linear, featuring high-degree polynomials with many coefficients.

Regardless of the camera model, one can use the techniques in this guide to calibrate a single camera.

Motivation, Objectives, & Background

There are many existing tools one can use to calibrate a camera. These exist both as standalone programs and SDK libraries. Given this, why would you want to write your own camera calibration module? There are some real reasons to do so: for instance, the existing tools may not support the hardware platform or parametric camera models you want to use.

Aside from any practical motivations, it's useful to work through the process of camera calibration to gain a better understanding of what's happening when you use a tool like OpenCV.

The full codebase for this tutorial can be found at the Tangram Visions Blog Repository.

*Using Rust!

This is especially useful when trying to understand why one of these tools may not be producing a suitable calibration.

Camera calibration is built on a foundation of linear algebra, 3D geometry and non-linear least squares optimization. One need not be an expert in all of these areas, but a basic understanding is necessary. To keep the blog post a reasonable length, some knowledge of linear algebra, the construction and use of 3D transformations, multivariable calculus and optimization (being able to understand a cost function should suffice) techniques is assumed for the result of the article.

The code snippets will be in Rust and we'll be using the NAlgebra linear algebra crate and the argmin optimization crate. The entire example can be found here.

Image Formation

To begin, we need a mathematical model of the formation of images. This provides a compact formula which approximately describes the relationship of 3D points in the scene to their corresponding 2D pixels in the image, aka *image formation or projection*. Projection is generally thought of in the following way:

- Light in a 3D scene hits the various objects therein and scatters.
- Some of that scattered light enters the camera's aperture.
- If a lens is present, the lens will refract (i.e. redirect) the light rays directing them to various pixels which measure the light incident to them during the exposure period.
-

One such model is called the pinhole model which describes image formation for pinhole cameras and camerae obscurae. We will use this model for this post due to its simplicity.

The pinhole model can work for cameras with simple lenses provided they're highly rectilinear, but most cameras exhibit some sort of lens distortion which isn't captured in a model like this. It's worth noting that most camera models ignore a lot of the real-life complexity of a lens (focus, refraction, etc).

Pinhole Background

In the pinhole model, the camera is said to be at the origin of a camera coordinate system shown in the diagram below. To image a 3D point, you draw a line between the point in question and the camera and see where that line intersects the virtual image plane, which is a model stand-in for the camera's physical sensor. This *intersection point* is then shifted and scaled according to the sensor's resolution into a pixel location. In the model, the image plane is set in front of the camera (i.e. along the +Z axis) at a distance called the *focal length*.

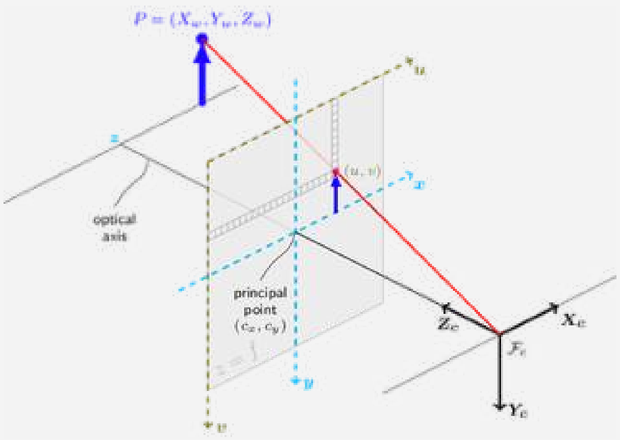


Figure 1: Pinhole camera model. Credit: OpenCV

The trick to deriving the formula for the intersection point from this diagram is to consider just two dimensions at a time. Doing this exposes a similar triangles relationship between points on the plane and points in the scene.

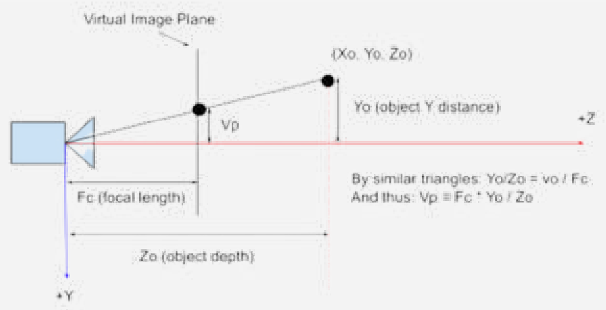


Figure 2: Pinhole model viewed from the side.

So we end up with the formula:

$$\begin{bmatrix} u_{plane} \\ v_{plane} \end{bmatrix} = \begin{bmatrix} F_c \frac{X}{Z} \\ F_c \frac{Y}{Z} \end{bmatrix}$$

Which maps points in 3D to the virtual image plane.

Getting To Pixels

The image plane is not an entirely abstract concept. It represents the actual CMOS sensor or piece of film of the camera. The intersection point is a point on this physical sensor; u_{plane} and v_{plane} are thus described in units of distance from a point on this sensor (e.g. meters). A further transformation which maps this plane location to actual pixels is required.

To do so, we need a few pieces of information:

- The *pixel pitch*: the edge length of a pixel given in meters-per-pixel (often micrometers-per-pixel)
- The *principal point*: the pixel location that any point on the camera's Z-axis maps to. The principal point (c_x, c_y) is usually in the center of the image (e.g. at pixel 320, 240 for a 640x480 image) but often deviates slightly because of small imperfections.

The principal point accounts for the discrepancy between the pixel-space origin and the intersection point. An intersection point on the Z axis will have zero X and Y components and thus projects to (u_{plane} , v_{plane}) = (0,0). Meanwhile, the origin in pixel-space is the upper left hand corner of the sensor.

After mapping to pixel space, we get our final pinhole camera model:

$$\begin{bmatrix} u_{pix} \\ v_{pix} \end{bmatrix} = \begin{bmatrix} \frac{f_c}{pp}\frac{X}{Z} + c_x \\ \frac{f_c}{pp}\frac{Y}{Z} + c_y \end{bmatrix}$$

It's common in computer vision parlance to group the focal length and pixel pitch terms into one pixel-unit focal length term since they're usually fixed for a given camera (assuming the lens cannot zoom). It's also very common to see distinct focal length parameters for the X and Y dimensions (f_x, f_y). Historically, this was to account for non-square pixels or for exotic lenses (e.g. anamorphic lenses), but in many scenarios having two focal length parameters isn't well-motivated and can even be detrimental. This results in the pinhole model in its most common form:

$$\begin{bmatrix} u_{pix} \\ v_{pix} \end{bmatrix} = P(\bar{X}; \{f_x, f_y, c_x, c_y\}) = \begin{bmatrix} f_x\frac{X}{Z} + c_x \\ f_y\frac{Y}{Z} + c_y \end{bmatrix}$$

It is the coefficients f_x, f_y, c_x, c_Y that we hope to estimate using camera calibration, as we will show in the next few sections of this guide.

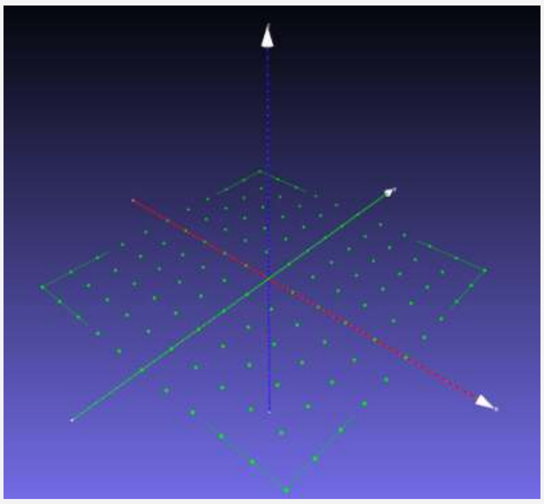
```
fn project(
    params: &na::Vector4<f64>, /*fx, fy, cx, cy*/
    pt: &na::Point3<f64>,
) -> na::Point2<f64> {
    na::Point2::<f64>::new(
        params[0] * pt.x / pt.z + params[2], // fx * x / z + cx
        params[1] * pt.y / pt.z + params[3], // fy * y / z + cy
    )
}
```

Zhang's Method

Many camera calibration algorithms available these days are some variant of a method attributed to Zhengyou Zhang (A Flexible New Technique for Camera Calibration by Zhengyou Zhang). If you've calibrated with a checkerboard using OpenCV, this is the algorithm you're using.

Put briefly, Zhang's method involves using several images of a planar target in various orientations relative to the camera to estimate the camera parameters. The planar target has some sort of automatically-detectable features that provide known data correspondences for the calibration system. These are usually grids of checkerboard squares or small circles. The planar target has known geometry and thus the point features on the target are described in some model or object coordinate system; these are often points on the XY plane, but that's an arbitrary convention.

Detecting and localizing these features is a topic in itself. We won't be covering it here.



An example planar target with points in a model coordinate system .

From Object to Image Coordinates

Before getting into the algorithm, it's useful to briefly revisit image formation. Recall that in order to apply a projection model, we need to be in the camera's coordinate system (i.e. camera at the origin, typically camera looks down the +Z axis etc.). We only know the target's dimensions in its coordinate system; we don't know precisely where the target is relative to the camera. Let's say that the there is a rigid body transform that maps points in the model coordinate system to the camera coordinate system: Γ_{model}^{camera} We can then write out the image formation equation for a given model point:

$$m_i = P(\Gamma_{model}^{camera} \cdot M_i; \{\theta_k\})$$

Where:

- m_i is the ith imaged model point (a 2-Vector in pixels)
- $m_i = P(\Gamma_{model}^{camera} \cdot M_i; \{\theta_k\})$ is the projection model, with *k* parameters$\{\theta_k\}$. This will be the pinhole model in further examples.
- $\Gamma_{model}^{camera} \cdot M_i$ is the ith model point transformed into the camera's coordinate frame.

We don't know this transform and moreover, we need one such transform for each image in the calibration data-set because either the camera or checkerboard or both can and should move between image captures. Thus we'll have to add these transforms to the list of things to estimate.

Cost Function

Ultimately the algorithm boils down to a non-linear least squares error minimization task. The cost function is based around reprojection error which broadly refers to the error between an observation and the result of passing the corresponding model point through the projection model. This is calculated for every combination of model point and image. Assuming we have good data and the model is a good fit for our camera, if we've estimated the parameters and transforms correctly, we should get near-zero error.

$$\{\theta_k\}, \{\Gamma_{model}^{camera_i}\} = \underset{\{\theta_k\}, \{\Gamma_{model}^{camera_i}\}}{\operatorname{argmin}} \sum_i^m \sum_j^n ||P(\Gamma_{model}^{camera_i} \cdot M_i; \{\theta_k\}) - m_i||_2^2$$

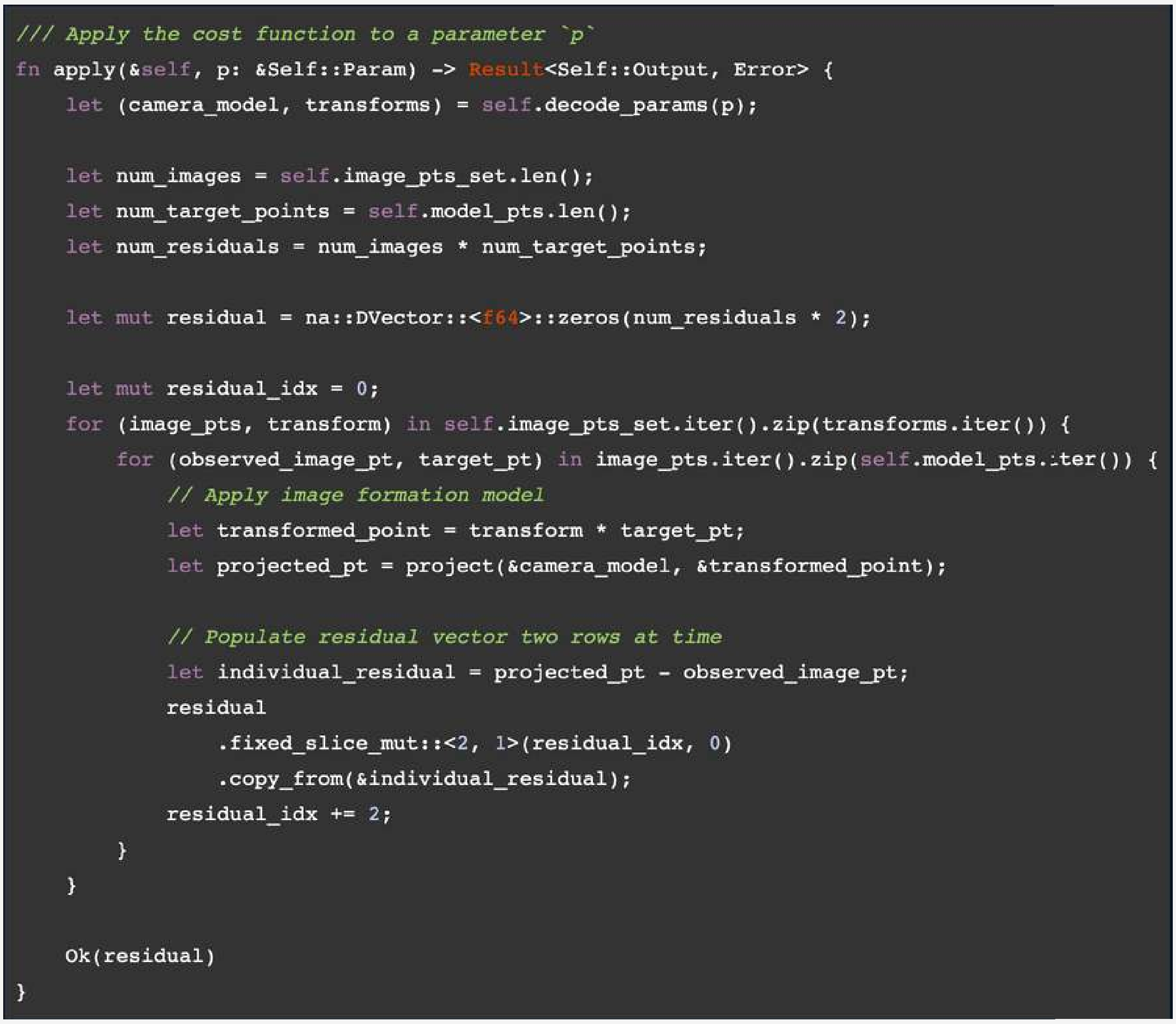
Non-Linear Least Squares Minimization

It should be clear from the above cost function that this is a least squares problem. In general, the cost function will also be nonlinear, meaning there won't be a closed form solution and we'll have to rely on iterative methods. There's a great variety of nonlinear least squares solvers out there. Among them are: Gauss-Newton (a good place to start if you want to implement one of these yourself), Levenberg-Marquardt, and Powell's dog leg method. There are a number of libraries that can implement these algorithms. For any of these algorithms to work you must provide them with (at a minimum) two things: the residual (i.e. the expression inside the $||\cdot||_2^2$ part of the cost function, and the Jacobian (matrix of first partial derivatives) of the residual.

Before we get into calculating these quantities, it's useful to reshape the cost function so that we have one large residual vector. The trick is to see that taking the sum of the norm-squared of many small residual vectors is the same as stacking those vectors into one large vector and taking the norm-squared.

$$\sum_i^m \sum_j^n ||r_{ij}||_2^2 = ||\begin{bmatrix} r_{00} & \cdots & r_{0m} & r_{10} & \cdots & r_{nm} \end{bmatrix}^T||_2^2$$

Calculating the residual is relatively straightforward. You simply apply the projection model and then subtract the observation, stacking the results into one large vector. We'll demonstrate calculating the residual in the code block below:



```
/// Apply the cost function to a parameter `p`
fn apply(&self, p: &Self::Param) -> Result<Self::Output, Error> {
    let (camera_model, transforms) = self.decode_params(p);

    let num_images = self.image_pts_set.len();
    let num_target_points = self.model_pts.len();
    let num_residuals = num_images * num_target_points;

    let mut residual = na::DVector::<f64>::zeros(num_residuals * 2);

    let mut residual_idx = 0;
    for (image_pts, transform) in self.image_pts_set.iter().zip(transforms.iter()) {
        for (observed_image_pt, target_pt) in image_pts.iter().zip(self.model_pts.iter()) {
            // Apply image formation model
            let transformed_point = transform * target_pt;
            let projected_pt = project(&camera_model, &transformed_point);

            // Populate residual vector two rows at time
            let individual_residual = projected_pt - observed_image_pt;
            residual
                .fixed_slice_mut::<2, 1>(residual_idx, 0)
                .copy_from(&individual_residual);
            residual_idx += 2;
        }
    }

    Ok(residual)
}
```

The Jacobian

The Jacobian is a bit more complicated. In this scenario, we're interested in the Jacobian of the stacked residual vector with-respect-to the unknown parameters that we're optimizing. By convention this Jacobian will have the following dimensions: $J \in \mathbb{R}^{2NM \times (6M+4)}$. The number of rows is the dimension of the output of the function which is 2NM because we have NM residuals each of which is 2D (because they're pixel locations). The number of columns is the number of parameters we're estimating and has two parts. The 4 represents the four camera parameters we're estimating. This number will change with the camera model. The 6M for M images represents the camera-from-model transforms describing the relative orientation of the camera and planar target in each image. A 3D transformation has six degrees of freedom.

The Jacobian can be thought of as being built up from "residual blocks". A residual block is one residual's contribution to the Jacobian and in this case a residual block is a pair of rows (pair because the residual is 2D). While the camera model is part of every residual block, each residual block only has one relevant transform.

This results in the Jacobian having a sparse block structure where, for a given residual block, only a few columns, those relating to the camera parameters and the transform at hand, will be non-zero. For larger calibration problems or least squares problems in general it can be beneficial to leverage the sparse structure to improve performance.

$$J = \begin{bmatrix} - & J_{r_{11uv}} & - \\ - & \vdots & - \\ - & J_{r_{nmuv}} & - \end{bmatrix} = \begin{bmatrix} J_{r_{11uv}\theta_k} & J_{r_{11uv}T_1} & \mathbf{0} & \cdots & \mathbf{0} \\ \vdots & & & & \\ J_{r_{n1uv}\theta_k} & J_{r_{n1uv}T_1} & \mathbf{0} & \cdots & \mathbf{0} \\ J_{r_{12uv}\theta_k} & \mathbf{0} & J_{r_{12uv}T_2} & \cdots & \mathbf{0} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ J_{r_{nmuv}\theta_k} & \mathbf{0} & \cdots & \mathbf{0} & J_{r_{nmuv}T_m} \end{bmatrix}$$

Each

- $J_{r_{ijuv}} \in \mathbb{R}^{2\times(6M+4)}$ is a pair of rows corresponding to the ith model point and jth image. It's made up of smaller Jacobians that pertain to the camera parameters or one of the transforms.
- $J_{r_{ijuv}\theta_k} \in \mathbb{R}^{2\times4}$ is the Jacobian of the residual function of the ith model point and jth image with respect to the four camera parameters.
- $J_{r_{ijuv}T_j} \in \mathbb{R}^{2\times6}$ is the Jacobian of the residual function of the ith model point and jth image with respect to the jth transform.
- $\mathbf{0} \in \mathbb{R}^{2\times6}$ is the Jacobian of the residual function of the any model point and jth image with respect to the any transform but the jth one. Only one transform appears in the residual equation at a time, so Jacobians for other transforms are always zero.

So now we know the coarse structure, but to fill out this matrix, we'll need to derive the smaller Jacobians.

Jacobian of the Projection Function

Let's define the residual function with respect to the parameters:

$$r(\theta_k) \in \mathbb{R}^2 = P(\Gamma_{model}^{camera_j} \cdot M_i; \{\theta_k\}) - m_i$$
$$\theta_k = [f_x, f_y, c_x, c_y]^T \in \mathbb{R}^4$$

We're looking for the Jacobian with the partial derivatives with-respect-to the model coefficients. Since there are four inputs and two outputs, we expect a Jacobian $J_{r\theta} \in \mathbb{R}^{2\times4}$.

$$J_{r\theta} = \begin{bmatrix} \frac{\delta P(X;\theta)}{\delta f_x} & \frac{\delta P(X;\theta)}{\delta f_y} & \frac{\delta P(X;\theta)}{\delta c_x} & \frac{\delta P(X;\theta)}{\delta c_y} \end{bmatrix} = \begin{bmatrix} \frac{x}{z} & 0 & 1 & 0 \\ 0 & \frac{y}{z} & 0 & 1 \end{bmatrix}$$
$$\text{where } X = \begin{bmatrix} x & y & z \end{bmatrix}^T = \Gamma_{model}^{camera_j} \cdot M_i$$

```
fn proj_jacobian_wrt_params(transformed_pt: &na::Point3<f64>)
-> na::Matrix2x4<f64> {
    na::Matrix2x4::<f64>::new(
        transformed_pt.x / transformed_pt.z,
        0.0,
        1.0,
        0.0,
        0.0,
        transformed_pt.y / transformed_pt.z,
        0.0,
        1.0,
    )
}
```

Jacobian of the Transformations

Because of the complexity of this topic, we're necessarily streamlining a few details for the sake of brevity. There's a broad topic in computer vision, computer graphics, physics, etc. of how to represent rigid body transforms. Common representations are: 4x3 matrices used with homogeneous points, separate rotation matrix and translation vector pairs, quaternions and translation pairs and Euler angles and translation pairs. Each of these have benefits and drawbacks but most of them are difficult to use in optimization problems. The main reason is that most of the rotation representations have additional constraints. For example, a rotation matrix must be orthogonal and a quaternion must be a unit quaternion. It's difficult to encode these constraints in the optimization problem without adding a lot of complexity. Without the constraints the optimization is free to change the individual elements of the rotation such that the constraint is no longer maintained. For example, with a rotation matrix representation, you may end up learning more general non-rigid linear transformation that can stretch, skew, scale etc.

For this optimization we're going to represent the transformations as 6D elements of a "Lie Algebra". Unlike the other representations, all values of this 6D vector are valid rigid-body transforms. I'm going to defer to other texts to explain this topic, as they can go pretty deep. The texts explain how to convert to (an operation called the "log map") a "lie algebra" element and convert from (an operation called the “exponetial map”) a “lie algebra”, as well as some Jacobians:

- A micro Lie theory for state estimation in robotics by Solà et al
- A tutorial on SE(3) transformation parameterizations and on-manifold optimization by Blanco-Claraco

We’ll need the Jacobian with the partial derivatives with-respect-to the six transform parameters. Since there are six inputs and two outputs, we expect a Jacobian $J_{rT} \in \mathbb{R}^{2\times 6}$.

Since we’re observing the effects of the transformation through the projection function we’ll have to employ the Chain rule:

$$J_{r\Gamma} = \frac{\delta r}{\delta \Gamma} = \frac{\delta P(\Gamma \cdot M)}{\delta \Gamma} = \frac{\delta P(X)}{\delta X}|_{X=\Gamma \cdot M} \cdot \frac{\delta \Gamma \cdot M}{\delta \Gamma}$$

So our Jacobian is written as a product of two other Jacobians. The first term is the matrix of partial derivatives of the projection function with-respect-to the point-being-projected (evaluated at the transformed point)

$J_{PX} \in \mathbb{R}^{2\times 3}$

$$J_{PX} = \begin{bmatrix} \frac{\delta P(X)}{\delta x} & \frac{\delta P(X)}{\delta y} & \frac{\delta P(X)}{\delta z} \end{bmatrix} = \begin{bmatrix} \frac{f_x}{z} & 0 & -\frac{f_x x}{z^2} \\ 0 & \frac{f_y}{z} & -\frac{f_y y}{z^2} \end{bmatrix}$$
$$\text{where } X = \begin{bmatrix} x & y & z \end{bmatrix}^T = \Gamma_{model}^{camera_j} \cdot M_i$$

```rust
fn proj_jacobian_wrt_point(
    camera_model: &na::Vector4<f64>, /*fx, fy, cx, cy*/
    transformed_pt: &na::Point3<f64>,
) -> na::Matrix2x3<f64> {
    na::Matrix2x3::<f64>::new(
        camera_model[0] / transformed_pt.z,
        0.0,
        -transformed_pt.x * camera_model[0] / (transformed_pt.z.powi(2)),
        0.0,
        camera_model[1] / transformed_pt.z,
        -transformed_pt.y * camera_model[1] / (transformed_pt.z.powi(2)),
    )
}
```

The second term is the matrix of partial derivatives of a point being transformed with-respect-to the transform:

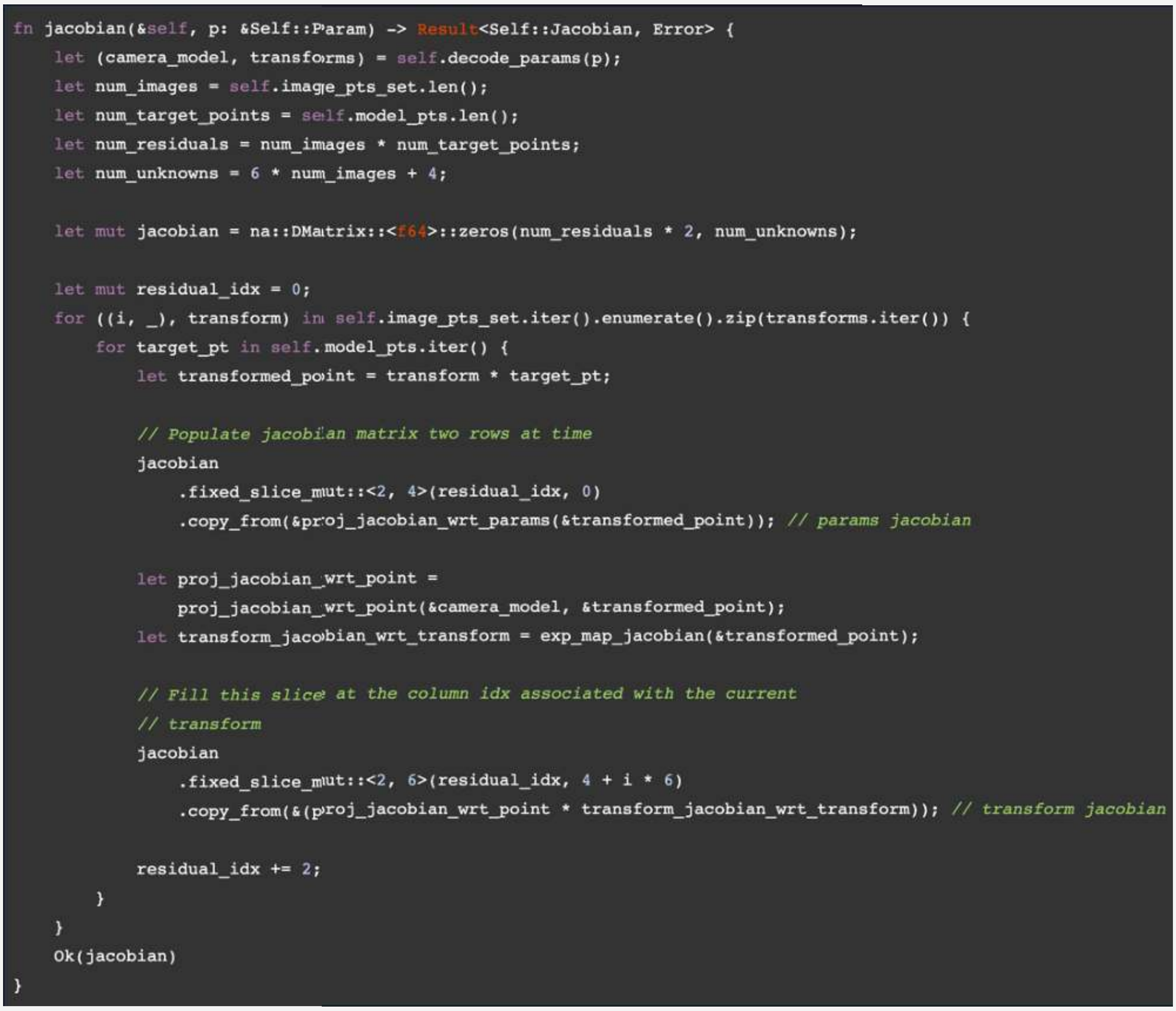
$$J_{T \cdot M} \in \mathbb{R}^{3\times 6}$$

We'll refer to the tutorial paper on how to derive this:

$$J_{T \cdot M} = \begin{bmatrix} 1 & 0 & 0 & 0 & -z & y \\ 0 & 1 & 0 & z & 0 & -x \\ 0 & 0 & 1 & -y & x & 0 \end{bmatrix}$$
$$\text{where } X = \begin{bmatrix} x & y & z \end{bmatrix}^T = \Gamma_{model}^{camera_j} \cdot M_i$$

```
fn skew_sym(v: na::Vector3<f64>) -> na::Matrix3<f64> {
    let mut ss = na::Matrix3::zeros();
    ss[(0, 1)] = -v[2];
    ss[(0, 2)] = v[1];
    ss[(1, 0)] = v[2];
    ss[(1, 2)] = -v[0];
    ss[(2, 0)] = -v[1];
    ss[(2, 1)] = v[0];
    ss
}

fn exp_map_jacobian(transformed_point: &na::Point3<f64>) -> na::Matrix3x6<f64> {
    let mut ss = na::Matrix3x6::zeros();
    ss.fixed_slice_mut::<3, 3>(0, 0)
        .copy_from(&na::Matrix3::<f64>::identity());
    ss.fixed_slice_mut::<3, 3>(0, 3)
        .copy_from(&(-skew_sym(transformed_point.coords)));
    ss
}
```

Building The Big Jacobian
We now have all the pieces we need to build the complete Jacobian. We do so two rows at a time for each residual, evaluating the various smaller Jacobians as described above. It's important to take care to populate the correct columns.



```
fn jacobian(&self, p: &Self::Param) -> Result<Self::Jacobian, Error> {
    let (camera_model, transforms) = self.decode_params(p);
    let num_images = self.image_pts_set.len();
    let num_target_points = self.model_pts.len();
    let num_residuals = num_images * num_target_points;
    let num_unknowns = 6 * num_images + 4;

    let mut jacobian = na::DMatrix::<f64>::zeros(num_residuals * 2, num_unknowns);

    let mut residual_idx = 0;
    for ((i, _), transform) in self.image_pts_set.iter().enumerate().zip(transforms.iter()) {
        for target_pt in self.model_pts.iter() {
            let transformed_point = transform * target_pt;

            // Populate jacobian matrix two rows at time
            jacobian
                .fixed_slice_mut::<2, 4>(residual_idx, 0)
                .copy_from(&proj_jacobian_wrt_params(&transformed_point)); // params jacobian

            let proj_jacobian_wrt_point =
                proj_jacobian_wrt_point(&camera_model, &transformed_point);
            let transform_jacobian_wrt_transform = exp_map_jacobian(&transformed_point);

            // Fill this slice at the column idx associated with the current
            // transform
            jacobian
                .fixed_slice_mut::<2, 6>(residual_idx, 4 + i * 6)
                .copy_from(&(proj_jacobian_wrt_point * transform_jacobian_wrt_transform)); // transform jacobian

            residual_idx += 2;
        }
    }
    Ok(jacobian)
}
```

Running the Algorithm
Practically speaking, Zhang's method involves iteratively minimizing the cost function, usually using a library. This means evaluating the residual and Jacobian with the current estimate of the transforms and camera parameters at each iteration until the cost function (hopefully) converges.

There are a few other practical concerns. The algorithm typically requires at least three images to get a good result. Additionally, the algorithm expects some variation in the camera-target orientation. Specifically, the algorithm performs better when the target is rotated away from the camera such that there's some variation in the depth of the target points. The original paper has further discussion about this.

If you're calibrating a camera model that has distortion parameters, it's also important to get enough images to uniformly cover all areas of the image plane so that the distortion function is fit correctly.

Given that this is an iterative algorithm, it is somewhat sensitive to the initial guess of the parameters. The original paper has some closed-form approximations which can provide good initial guesses. There seems to be a relatively large space of parameters and transforms which still converge to the right answer. A ballpark guess of the parameters and some reasonable set of transforms (e.g. ones that place the target somewhere directly in front of the camera) is often good enough.

Putting It Together In Rust

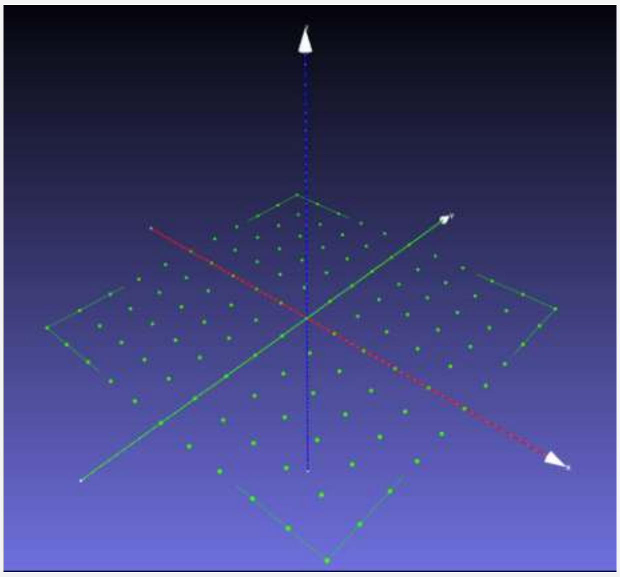
Because the Tangram Vision Platform is written in Rust (with wrappers for C and ROS, of course), we'll demonstrate how to create a calibration module in Rust. We'll be using the NAlgebra linear algebra crate and the argmin optimization crate. You can follow along here, or with the full codebase at the Tangram Visions Blog repository. Let's get started.

Generating Data

In this tutorial we aren't going to work with real images, but rather synthetically generate a few sets of image points using ground truth transforms and camera parameters.

To start we'll generate some model points. The planar target lies in the XY plane. It will be a meter on each edge.

```rust
let mut source_pts: Vec<na::Point3<f64>> = Vec::new();
for i in -5..6 {
    for j in -5..6 {
        source_pts.push(na::Point3::<f64>::new(i as f64 * 0.1, j as f64 * 0.1, 0.0));
    }
}
```


Model points in the model coordinate system (on the XY plane).

Then we'll generate a few arbitrary camera-from-model transforms and some camera parameters.

```
let camera_model = na::Vector4::<f64>::new(540.0, 540.0, 320.0, 240.0); // fx, fy, cx, cy
let transforms = vec![
    na::Isometry3::<f64>::new(
        na::Vector3::<f64>::new(-0.1, 0.1, 2.0),//translation
        na::Vector3::<f64>::new(-0.2, 0.2, 0.2),//rotation
    ),
    na::Isometry3::<f64>::new(
        na::Vector3::<f64>::new(-0.1, -0.1, 2.0),
        na::Vector3::<f64>::new(0.2, -0.2, 0.2),
    ),
    na::Isometry3::<f64>::new(
        na::Vector3::<f64>::new(0.1, 0.1, 2.0),
        na::Vector3::<f64>::new(-0.2, -0.2, -0.2),
    ),
];
```

Finally we'll generate the imaged points by applying the image formation model.

```
let transformed_pts = transforms
    .iter()
    .map(|t| source_pts.iter().map(|p| t * p).collect::<Vec<_>>())
    .collect::<Vec<_>>();

let imaged_pts = transformed_pts
    .iter()
    .map(|t_list| {
        t_list
            .iter()
            .map(|t| project(&camera_model, t))
            .collect::<Vec<na::Point2<f64>>>()
    })
    .collect::<Vec<_>>();
```

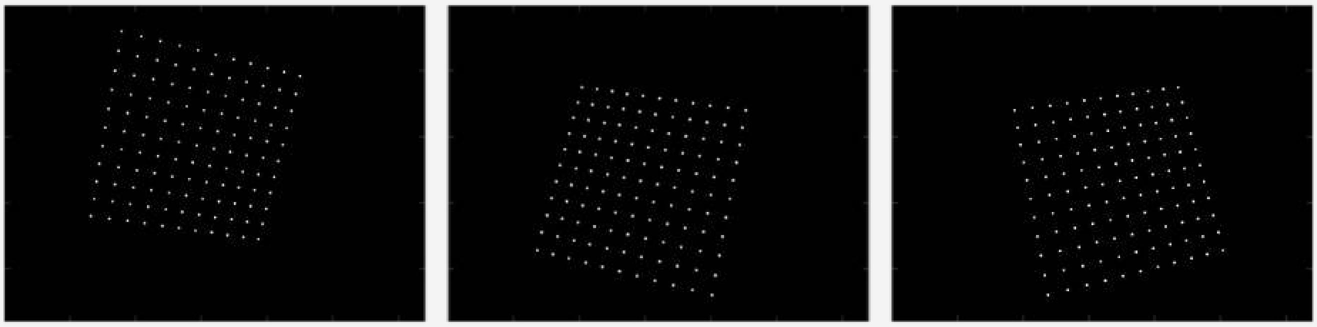
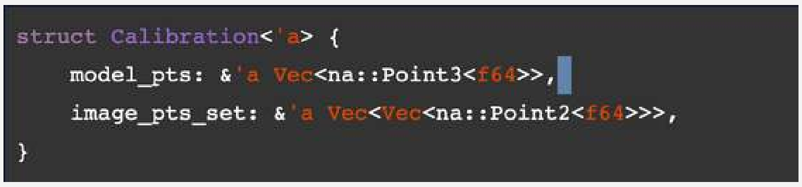


Figure 4: synthetic images rendered by applying the ground truth camera-from-model transform and then projecting the result using ground truth camera model.

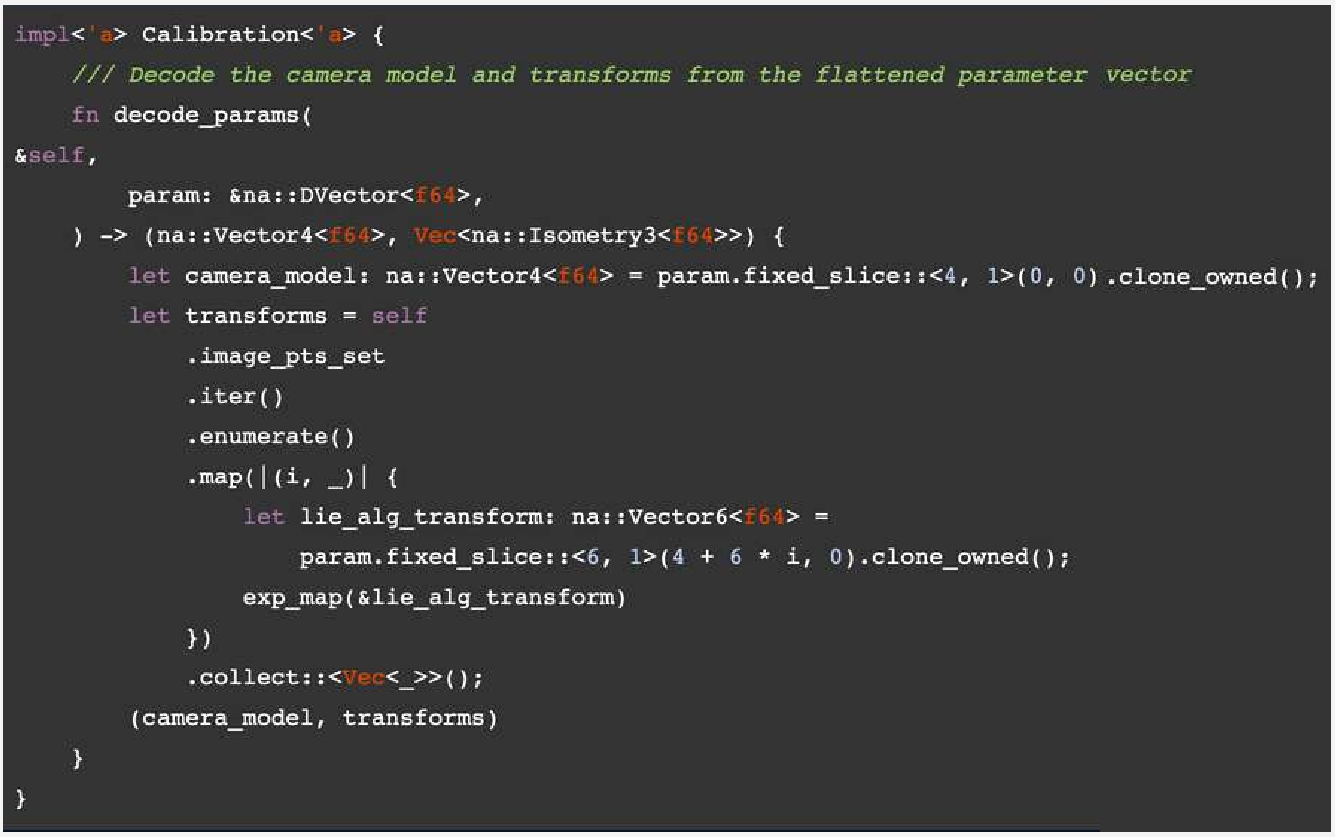
Building the Optimization Problem

We're going to use the argmin crate to solve the optimization problem. To build your own problem with argmin, you make a struct which implements the ArgminOp trait. The struct holds the data we're processing. Depending on the optimization algorithm you select, you'll have to implement some of the trait's functions. We're going to use the Gauss-Newton algorithm which requires that we implement apply() which calculates the residual vector and jacobian() which calculates the Jacobian. The implementations for each are in the previous sections.



```
struct Calibration<'a> {
    model_pts: &'a Vec<na::Point3<f64>>,
    image_pts_set: &'a Vec<Vec<na::Point2<f64>>>,
}
```

During the optimization problem, the solver will update the parameters. The parameters are stored in a flat vector and thus it's useful to make a function that converts the parameter vector into easily-used objects for calculating the residual and Jacobian in the next iteration. Here's how we've done it:



```
impl<'a> Calibration<'a> {
    /// Decode the camera model and transforms from the flattened parameter vector
    fn decode_params(
&self,
        param: &na::DVector<f64>,
    ) -> (na::Vector4<f64>, Vec<na::Isometry3<f64>>) {
        let camera_model: na::Vector4<f64> = param.fixed_slice::<4, 1>(0, 0).clone_owned();
        let transforms = self
            .image_pts_set
            .iter()
            .enumerate()
            .map(|(i, _)| {
                let lie_alg_transform: na::Vector6<f64> =
                    param.fixed_slice::<6, 1>(4 + 6 * i, 0).clone_owned();
                exp_map(&lie_alg_transform)
            })
            .collect::<Vec<_>>();
        (camera_model, transforms)
    }
}
```

Running the Optimization Problem

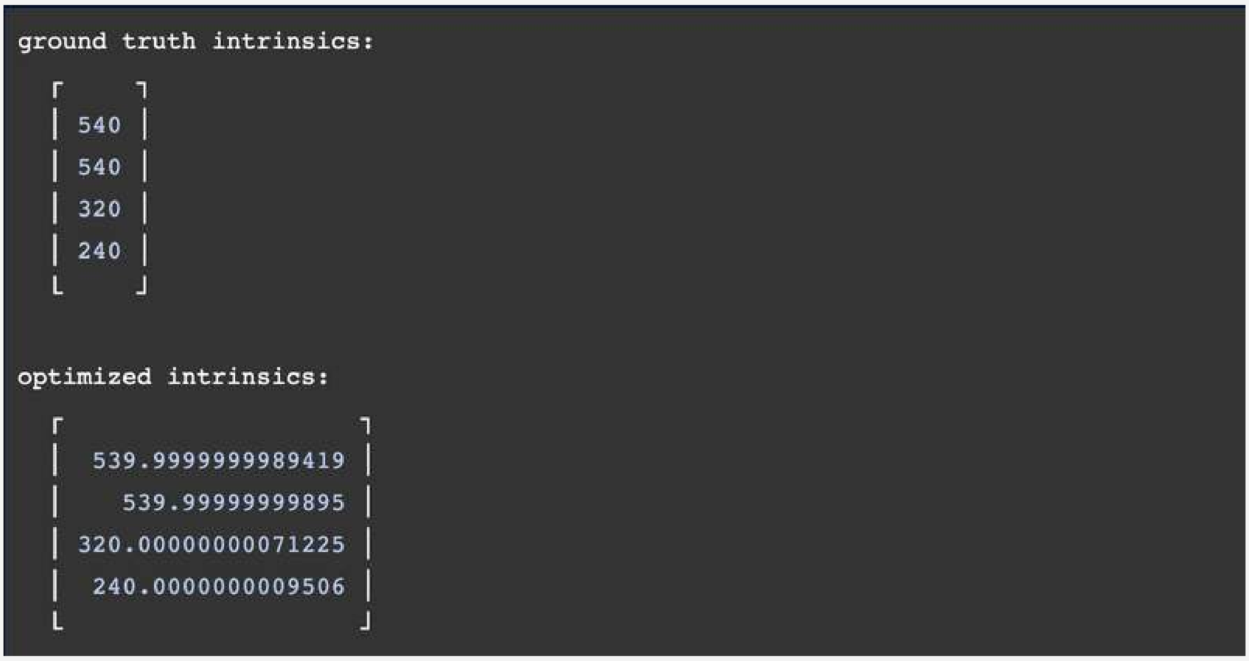
Before we run the optimization problem we'll have to initialize the parameters with some reasonable guesses. In the code block below, you'll see that we input four values for f_x, f_y, c_x, and c_y .

```rust
let mut init_param = na::DVector::<f64>::zeros(4 + imaged_pts.len() * 6);

// Arbitrary guess for camera model
init_param[0] = 1000.0; // fx
init_param[1] = 1000.0; // fy
init_param[2] = 500.0; // cx
init_param[3] = 500.0; // cy

// Arbitrary guess for poses (3m in front of the camera with no rotation)
let init_pose = log_map(&na::Isometry3::translation(0.0, 0.0, 3.0));

// Populate poses with guess
init_param
    .fixed_slice_mut::<6, 1>(4, 0)
    .copy_from(&init_pose);
init_param
    .fixed_slice_mut::<6, 1>(4 + 6, 0)
    .copy_from(&init_pose);
init_param
    .fixed_slice_mut::<6, 1>(4 + 6 * 2, 0)
    .copy_from(&init_pose);
```

Next we build the argmin "executor" by constructing a solver and passing in the ArgminOp struct.

```rust
let cal_cost = Calibration {
    model_pts: &source_pts,
    image_pts_set: &imaged_pts,
};
let solver = argmin::solver::gaussnewton::GaussNewton::new()
    .with_gamma(1e-1)
    .unwrap();
let res = Executor::new(cal_cost, solver, init_param)
    .add_observer(ArgminSlogLogger::term(), ObserverMode::Always)
    .max_iters(2000)
    .run()
    .unwrap();

eprintln!("{}\\n\\n", res);
eprintln!("ground truth intrinsics: {}", camera_model);
eprintln!(
    "optimized intrinsics: {}",
    res.state().best_param.fixed_slice::<4, 1>(0, 0)
);
```

After letting the module run for over 100 iterations, the cost function will be quite small and we can see that we converged to the right answer.



```
ground truth intrinsics:

  ┌     ┐
  │ 540 │
  │ 540 │
  │ 320 │
  │ 240 │
  └     ┘


optimized intrinsics:

  ┌                    ┐
  │  539.9999999989419 │
  │    539.99999999895 │
  │ 320.00000000071225 │
  │  240.0000000009506 │
  └                    ┘
```

In Conclusion

So there you have it - the theory and the execution of a built-from-scratch calibration module for a single sensor.

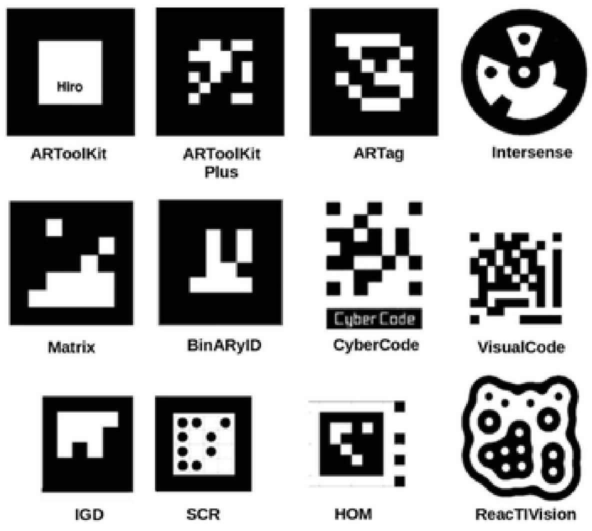
Creating a calibration module for sensors is no trivial task. The simplified module and execution we've described in these two posts provide the building blocks for a single camera approach, without optimizations for calibration time, compute resources, or environmental variability. Adding these factors, and expanding calibration routines to multiple sensors, makes the calibration challenge significantly more difficult to achieve. That's why multi-sensor, multi-modal calibration is a key part of the Tangram Vision Platform.

ARUCO, CHARUCO, AND APRILTAGS: CREATING FIDUCIAL MARKERS FOR VISION

If you've been in or around computer vision for a while, you might have seen one of these things:

These are known as *fiducial markers*, and are used a way to establish a visual reference in the scene. They're easy to make and easy to use. When used in the right context, extracting these markers from a scene can aid in camera calibration, localization, tracking, mapping, object detection... almost anything that uses camera geometry, really.

What you might not know is that there are a *lot of different fiducial markers*:



Different fiducial markers. From http://cs-courses.mines.edu/csci507/schedule/24/ArUco.pdf

Some of these are older, some are newer, some are "better" in a given scenario. However, all of them share the same very clever mathematical foundation. A few smart yet straightforward ideas allow these fiducial markers to perform well under different lighting conditions, upside-down, and in even the harshest environment: the real world.

In my opinion, the best way to understand this math magic is to try it ourselves. What goes into making a robust fiducial marker? Can we pull it off? We'll use the ArUco tag design as our reference. The ArUco tag is one of the most common fiducial markers in computer vision, and it checks a lot of boxes as far as robustness and usability goes. Let's reverse-engineer ArUco markers and discover why these work so well.

ArUco (also stylized ARUCO or Aruco) got its moniker from Augmented Reality, University of Cordoba, which is the home of the lab that created the marker. At least, that's what it seems to be; I've never seen it documented!

What To Look For

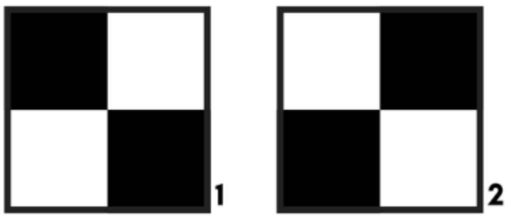
As a rule of thumb, our marker shouldn't look like anything found in natural images:

- No color gradients
- Sharp, consistent angles
- Consistent material

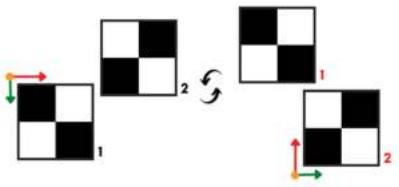
It doesn't take much brainstorming to realize that "black-and-white matte square" fits a lot of these qualifications. In addition, square corners and edges are each themselves unique features, giving us more bang for our buck. Let's use this concept as our base.

It's easy enough to track one black-and-white square frame to frame in a video (and some algorithms do!), but what if we want to use more than one marker? Having two identical squares makes it impossible to identify which is which. We clearly need some way to disambiguate.

If we want to maintain our not-from-nature design choices, keeping black-and-white squares around somehow makes sense. Why don't we use... combinations of squares?



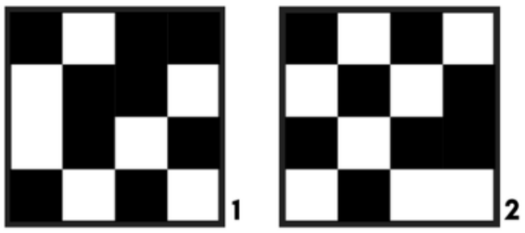
We can now differentiate, to some extent. But clever minds will notice that a rotation in our camera will confuse our identification:



Obviously, we'll have to do better than this.

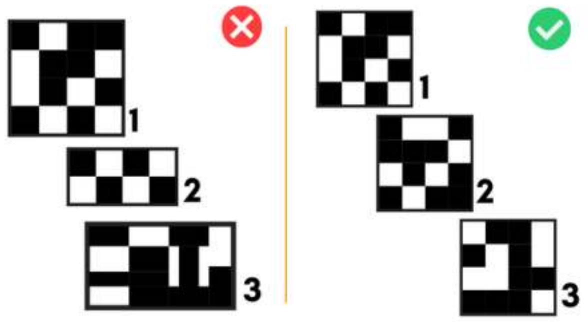
Squares of Squares

...so let's use more squares! We can switch from black to white across squares to up the entropy in our pattern.



This looks better, but even with more squares, we *still* have the risk of mixing it up with another marker design. It's clear that we want to avoid mis-classifying our markers, and the more differentiable our markers are, the easier it will be to detect each of them correctly. However, there's a balance to strike here: if our patterns are *too* different from one another, or don't have *enough* features, it won't be easy for our camera to tell that something is a marker at all!

This means that our squares should have a *common structure*, but *different* and *varied features*. The easiest way to add structure is to define a constant shape (in square rows and columns) of every marker that the camera might see. This specificity makes it clearer what our algorithm is trying to find, and sets conditions for identification once it finds it.



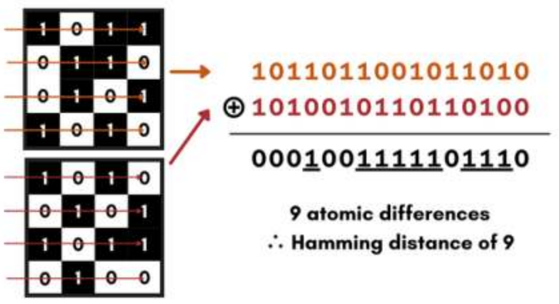
Now, with our structure well-defined, we can focus on the original problem: making every individual marker that fits this structure as different as possible from all of its kin.

Hamming It Up, Going the Distance, etc, etc

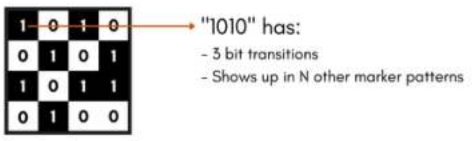
Let's put that another way: we want to maximize the difference between all of our marker patterns that share that same structure. Whenever the word "maximize" comes up in algorithmic design, one should immediately think of optimization. If we can somehow translate our marker patterns into a formula or algorithm, we can turn our predicament into an optimization problem which can be better understood and solved.

Markers like ArUco do this by treating every square in the pattern like a bit of information. Every pattern can therefore be represented by a bit string, and the difference between two patterns is the minimum number of changes it takes to turn the first into the second.

This is called the Hamming distance between markers, named after mathematician Richard Hamming. By representing the difference in Hamming distance, we can formulate our goal in terms of formulas, not just words.



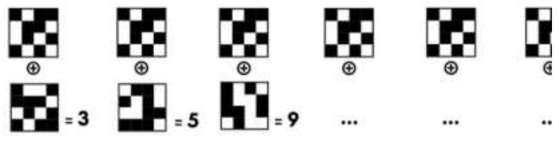
So, now that we can represent every marker as its corresponding bit pattern, we can make some math! We want to make sure that every marker has as many black-to-white transitions as possible...



$$P(w = w_i) = \frac{T(w_i)O(w_i, \mathfrak{D})}{\sum_{w_j \in W} T(w_j)O(w_j, \mathfrak{D})}$$

Probability of word generation: The probability that a bit word (represented as a row in the marker) will be added to the marker generated.
A function of the number of bit transitions in the word T(w) and the number of times that word appears in other markers O(w, D)

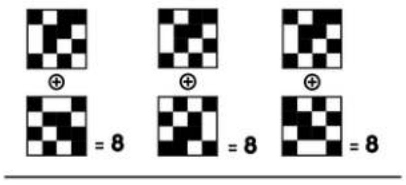
...while making its pattern as different as possible from every other marker...



$$D(m_i, \mathfrak{D}) = \min_{m_j \in \mathfrak{D}} \{D(m_i, m_j)\},$$

Distance of marker to dictionary: The minimum Hamming distance between a marker and every other marker that has been generated.

...while also making sure that this marker's rotations are unique from itself, so that we don't mix it up with another marker when our camera is tilted.



$$S(m_i) = \min_{k \in \{1,2,3\}} \{H(m_i, R_k(m_i))\}.$$

Marker self-distance: The minimum Hamming distance between a marker and its rotations.

Now use all of these formulas together to optimize every new marker you can generate. Collect those markers that hit a certain threshold, and do it again and again until you have as many as you need! We now have a complete dictionary of fiducial markers.

Richard Hamming's innovations around bit-wise information storage and transfer were a huge influence on the Information Age. Though we won't cover them here, 3Blue1Brown's wonderful explanation on the power and simplicity of Hamming Codes is worth a watch.

Hitting Our Limits

Now that we have our dictionary, we can start detecting. Our optimization process gave us some pretty cool abilities in this regard. For one, we have a much lower chance of mixing up our markers (which is what we optimized for). More surprisingly, though, we've also made ourselves more robust to detection mistakes.

For instance, what if we detect a white square in a marker, when in reality that same square was black? Since we've maximized the Hamming distance between all of our markers, this mistake shouldn't cost us. Instead, we'll select the matching marker in our dictionary that has the closest Hamming distance to our detected features. The misdetection was unfortunate, but we took it in stride.

What if we misdetect several squares? At some point, the misdetections will be too much for our poor dictionary, and we'll start getting confused. But when do we hit this threshold?

An easy way to set that breaking point ourselves is by adding a minimum Hamming distance constraint between all markers in our dictionary. If our minimum Hamming distance is 9, and we generate a marker that has a distance of 8 from the rest of the dictionary, we throw it out. This process limits the number of markers that we can add, but in exchange we generate a dictionary that's more robust to detection mistakes. This ability becomes more crucial as your marker size grows and the feature details become finer.

Small disclaimer: Fiducial marker libraries like AprilTag employ this strict Hamming distance threshold when generating their libraries. However, ArUco derives a maximum value to its cost functions instead (those formulas we derived above). The effect is the same: more robust dictionaries.

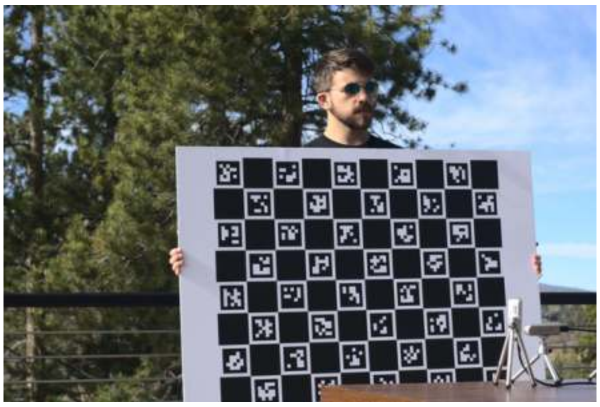
A Solid Fiducial Marker

There you have it: the basics of ArUco! All we needed to do was

- Create a varied, redundant pattern
- Fit it to a certain mathematical structure
- Generate as many markers as we can that fit this structure
- Throw out the markers that are too similar to any other marker, or variants of itself, or within a certain Hamming distance

Of course, there are small tweaks and changes one can make to this basic formula to up the robustness, or variety, or usefulness in different environments. This is where all those other markers come from! Regardless, these other markers all considered the same points we did today; they just arrived at different solutions.

Tangram Vision's calibration module can use several fiducial types to help bootstrap the camera calibration process. While we're no fan of the checkerboard (which we'll let you know), we all recognize the value of fiducials when used at the right place and time. Even your humble author has been known to wave around a few markers now and then:



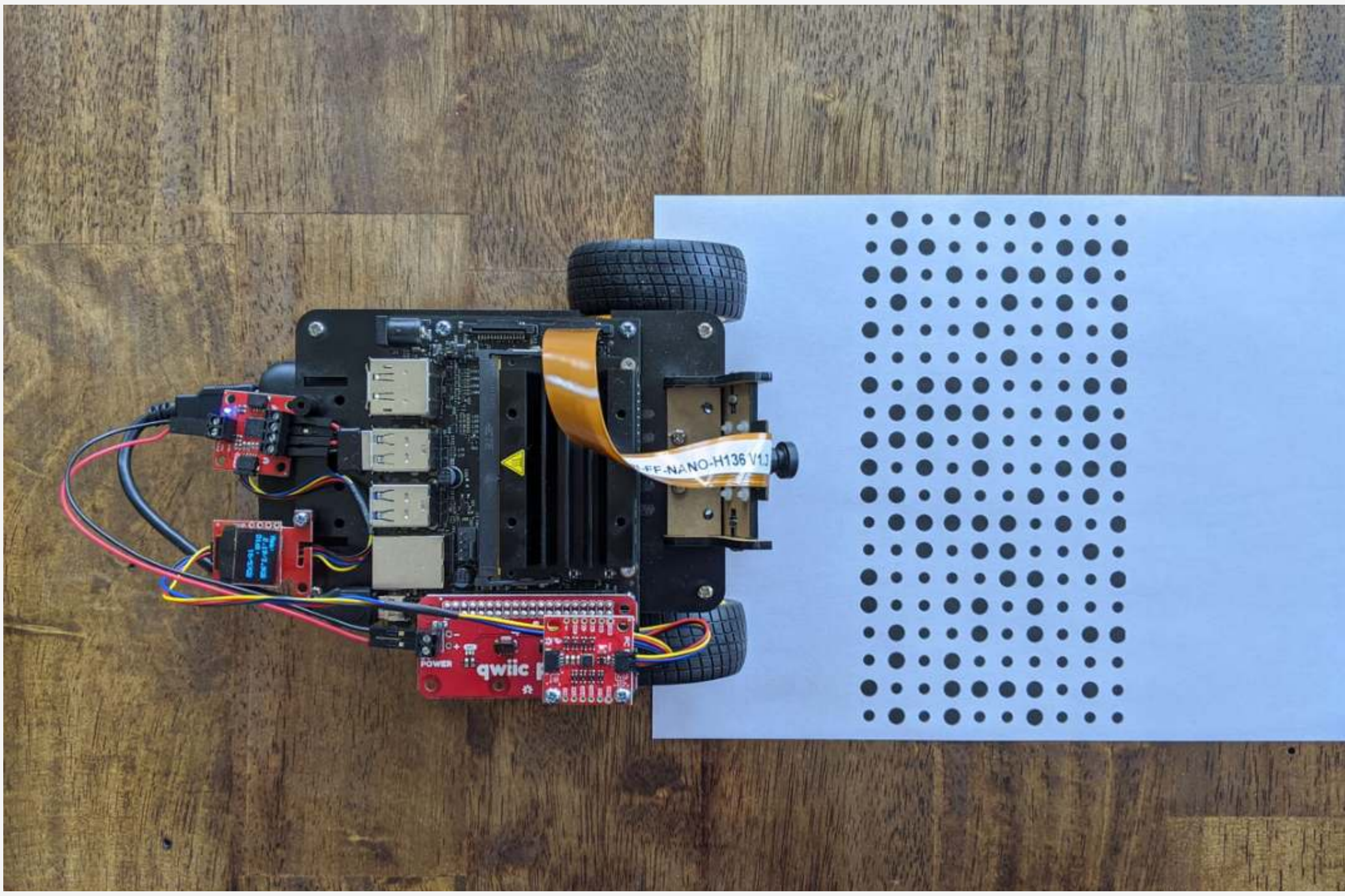
Further Reading

- Automatic generation and detection of highly reliable fiducial markers under occlusion. The original ArUco paper!
- ArUco Library Documentation. Docs on the software that's used to produce and detect ArUco targets. Written by the people who created it.
- Generation of fiducial marker dictionaries using mixed integer linear programming. A good overview of various fiducial optimization techniques, while also offering the authors' own takes on the process.

SECTION III: UNDERSTANDING CALIBRATION RESULTS

67 / Calibration Statistics: Accuracy Vs Precision

72 / Why Calibration Matters For AI/ML



CALIBRATION STATISTICS: ACCURACY VS PRECISION

One question that I have received time and time again in my career is

"*How accurate is your calibration process?*"

This always bothers me. If I answer "very accurate, of course!", what does that... mean? What should they expect from results? What am I even claiming?

Even more interesting: a question that I've never been asked, but that has equal or greater importance, is "How precise is this calibration?"

Accuracy is always considered, while precision is never questioned. What makes this so?
There are probably a few reasons for this:

1. Most people don't understand the distinction between accuracy and precision in general, much less in sensor calibration
2. It is not common practice to measure precision anyway

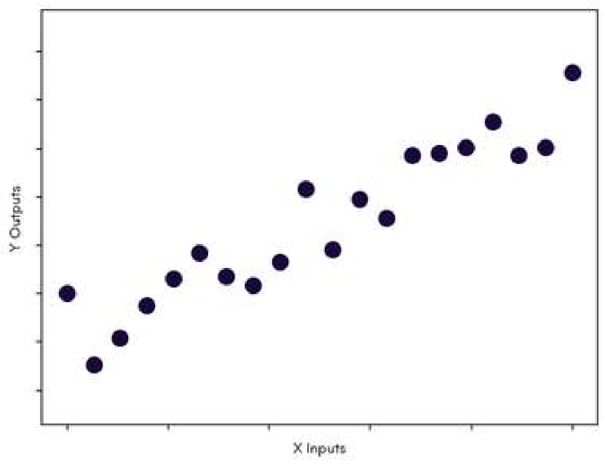
Both of these points are a shame; one should have a good understanding of accuracy and precision in order to get the most out of their perception systems, or any calibrated system for that matter. In order to do this, though, we need some context.

Caveat: Statistics is a very dense field of study. There's a lot of nuance here that won't get covered due to the inordinate amount of detail behind the subject. However, the thesis should hopefully be clear: most current calibration processes leave out some important stuff. If you're the type of person to dive deep into this kind of thing, and are looking for a career in perception...you know where to find us.

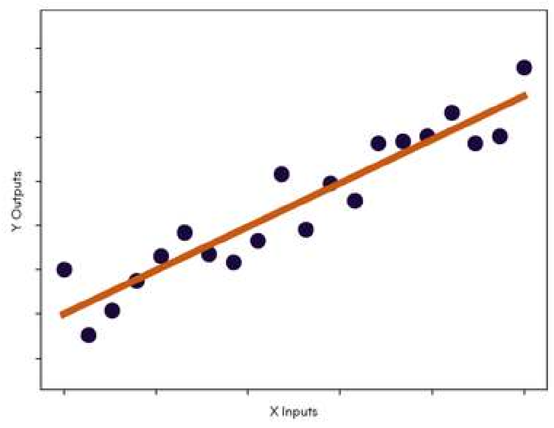
Accuracy and Precision in Model Fitting

Imagine that we have a set of points on a graph. We would like to **predict** the output value (the Y axis) of every input value (the X axis).

We do this because we are totally normal and cool and have fun in our free time.



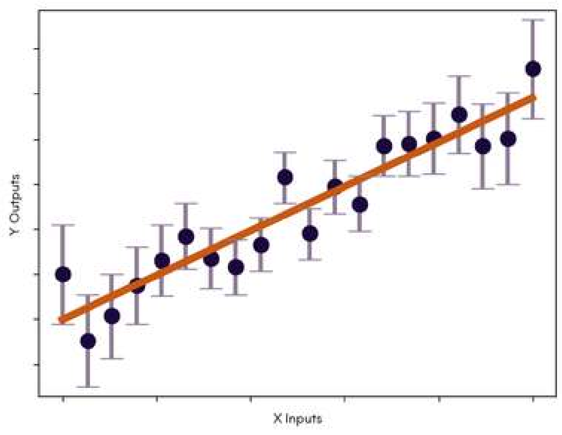
We can do this by fitting a model to the data. Good models give us a general idea of how data behaves. From the looks of it, these points seem to follow a straight line; a linear model would probably work here. Let's do this:



The input-output points that are on our graph are close to or on the line that we derived. This means that our linear model has a high **accuracy** given this data.

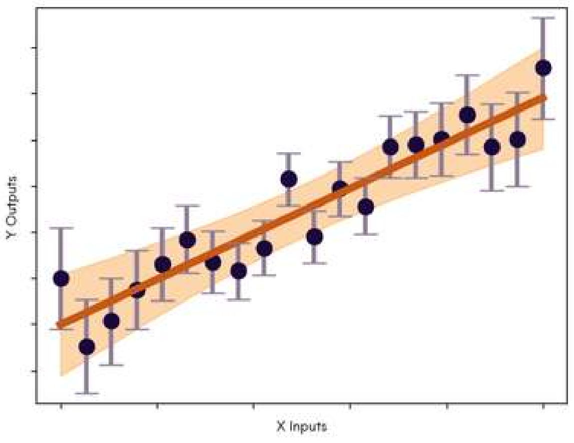
Yet there's something that's not shown in this graph. When we collected our datapoints, we used a tool that only measured so well. In other words, it was impossible to get exact measurements; we just used the best measurements we could get. This means that every datapoint has an uncertainty, or **variance**, associated with it. This is a representation of the **precision** of the input space.

When we factor in the input space variance, our graph of points really looks like this:



Now the linear model we derived looks a little less perfect, huh? Yet this is not reflected in our results at all. If every input point measurement was actually a bit different than what's shown, the true model for this data is something else entirely. All of our predictions are off.

The good news is that there are ways to compensate for this uncertainty. Through the magic of statistics, we can translate the input variance into variance of our learned model parameters:



This gives us a much richer understanding of our input-output space. Now we know that our model can be trusted much more with certain inputs than with others. If we're trying to predict in a region outside of our input data range, we have a good idea of how much we can trust (or be skeptical of) the data. This is called **extrapolation**; it's a hard thing to get right, but most people do it all the time without considering the precision of their model.

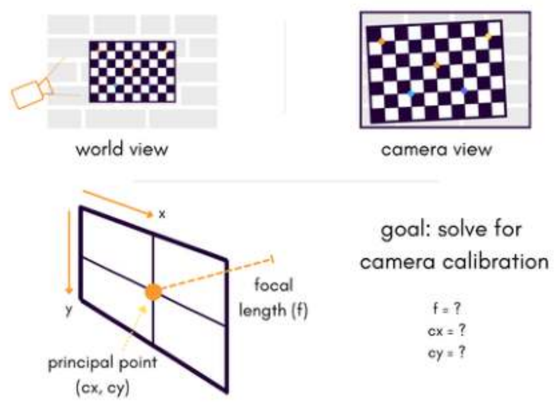
You can try this for yourself by playing around with the source code used to develop these graphs. Find it at the Tangram Visions Blog repository.

There's a strong moral here: accuracy isn't everything. Our model *looked* accurate, and given the input data, it was! However, it didn't tell the whole story. We needed the precision of our model to truly understand our predictive abilities.

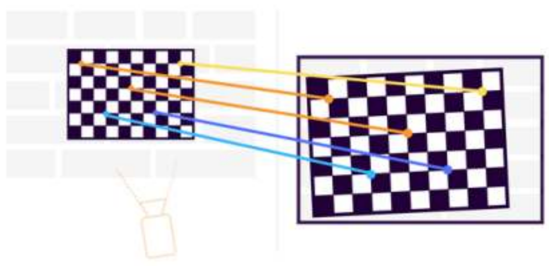
Accuracy in Camera Calibration

The same logic that we used in the line-fitting example applies directly to calibration. After all, calibration is about fitting a **model** to available data in order to predict the behavior of outputs from our inputs. We get better prediction with a better understanding of both the calibration accuracy and precision.

To demonstrate this, let's calibrate a camera using a target.



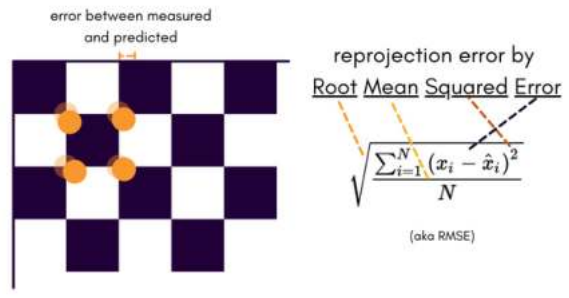
Our target provides metric data that we can use to connect the camera frame with the world frame; we call this metric data the object space. An image taken by the camera of the target captures a projection of the **object space** onto the image plane; this is called image space. We produce a new set of object space inputs to **image space** outputs with every image.



The object-to-image data in each camera frame act like the points in our linear fit example above. The camera model we refine with this data provides a prediction about how light bends through the camera's lens.

To dive into the mathematics of this process, see our blog post on camera calibration in Rust.

Just like in our graph, the accuracy of our derived camera model is a comparison of predicted and actual outputs. For camera calibration, this is commonly measured by the **reprojection error**, the distance in image space between the measured and predicted point.


error between measured and predicted
reprojection error by Root Mean Squared Error
$\sqrt{\frac{\sum_{i=1}^{N}(x_i - \hat{x}_i)^2}{N}}$
(aka RMSE)

For those who are familiar with camera calibration: It's important to note that reprojection error calculated on the input data alone doesn't tell the whole story. In order to actually get accuracy numbers for a calibration model, one should calculate the reprojection error again using data from outside of your training set. This is why some calibration processes put aside every other reading or so; it can use those readings to validate the model at the end of optimization.

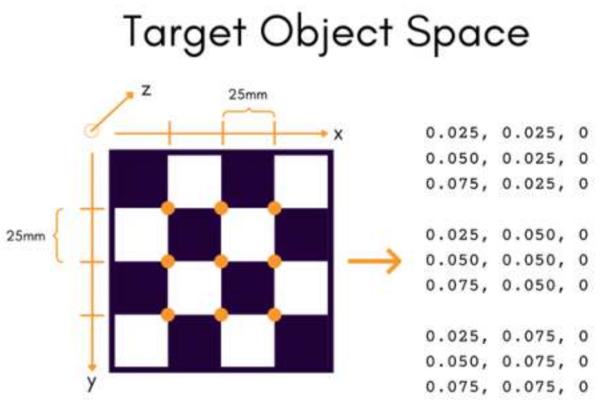
A common rule of thumb is if the reprojection root mean squared error (RMSE) is under 1 pixel, the camera is "calibrated". This is the point where most calibration systems stop. Does anything seem off with this?

Precision in Camera Calibration

The answer is yes! Red flags abound, and they all lie in our uncertainties. Except this time, it's more than just our inputs; in camera calibration, we have knowledge, and therefore uncertainty, about our parameters as well.

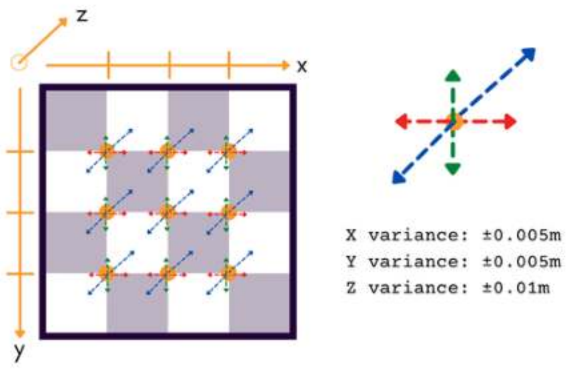
Object Space

Let's start with the object space, i.e. our input. Traditional calibration targets are mounted on a flat surface. This gives us metric points on a plane.



```
0.025, 0.025, 0
0.050, 0.025, 0
0.075, 0.025, 0

0.025, 0.050, 0
0.050, 0.050, 0
0.075, 0.050, 0

0.025, 0.075, 0
0.050, 0.075, 0
0.075, 0.075, 0
```

However, what happens when this board bends? This assumption of a planar metric space is no longer valid. Luckily, we can account for this through a variance value for each object space point.



```
X variance: ±0.005m
Y variance: ±0.005m
Z variance: ±0.01m
```

This is a step that very few calibration processes take into account. Unless the target is mounted on a perfectly flat surface like a pane of glass, there will be inconsistencies in the plane structure. This will lead to a calibration that is imprecise. However, given that the target points are taken as ground truth, the calibration model will give the illusion of accuracy. Whoops.

"Perfectly flat" here is relative. If the deviation from the plane is small enough that we can't observe it... it's flat enough.

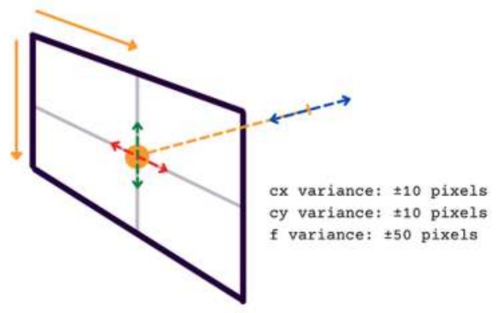
Parameter Space

In our linear model example above, we didn't know what real-world effect our line was modeling. Any guess at our parameter values before the line fit would have probably been way off. But what if we were confident in our initial guesses? How would we convey this information to our model fitting optimization?

Good news: variances do this, too! Assigning variances to our parameters tells the model optimization how certain we are of the initial guess. It's a straightforward way to tune the optimization, as we'll see later

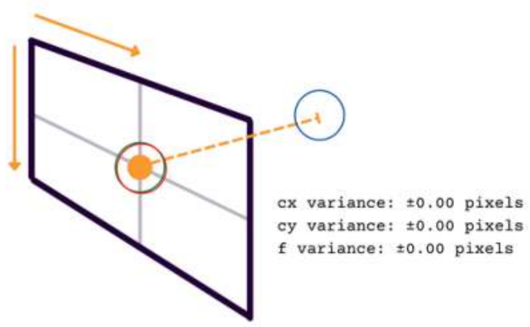
As an example applied to camera calibration: the principal point of a camera lens is often found around the center of the image. We're fairly certain that our lens will follow this paradigm, so we can assign a low variance to these parameters. If we have enough of an understanding of our model, we can do this for every parameter value.

Now our optimization process has a bit more information to work with; it knows that some parameters have more room to deviate from their initial guess than others.



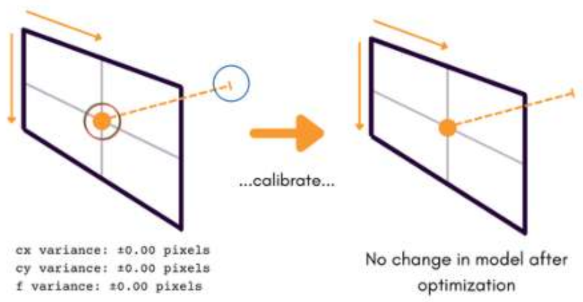
```
cx variance: ±10 pixels
cy variance: ±10 pixels
f variance: ±50 pixels
```

We can get a better idea of this effect by looking at its extreme. Let's assign all of our parameters a variance of 0.00, i.e. no uncertainty at all.



```
cx variance: ±0.00 pixels
cy variance: ±0.00 pixels
f variance: ±0.00 pixels
```

Now, when we calibrate over our object space with these parameter values, we find that... our model doesn't change! All of our parameters have been fixed.


...calibrate...
cx variance: ±0.00 pixels
cy variance: ±0.00 pixels
f variance: ±0.00 pixels
No change in model after optimization

This makes sense, since we indicated to the model optimization that we knew that our initial parameters had no *uncertainty* via the 0.00 variance value. Thus, a larger variance value allows the parameter value to move around more during optimization. If we're very *uncertain* about our guess, we should assign a large initial variance.

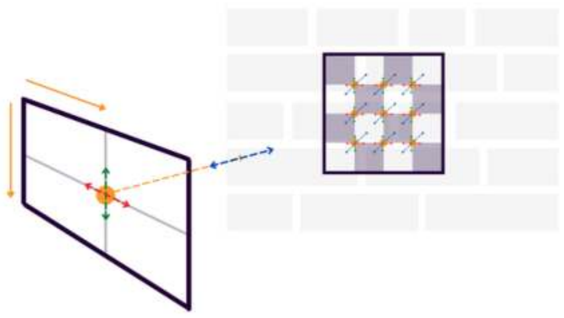
Our parameter variances will change as we fit our model to the input data. This becomes a good measure of model precision, just like in our linear fit example. We're now armed with enough knowledge to predict our model's behavior confidently in any scenario.

And yet...

With all of that said, when is the last time you measured the precision in your calibration process? Odds are that the answer is "never". Most calibration systems don't include these factors! That's a *huge* problem; as we've seen, accuracy won't paint the whole picture. Factoring in precision gives

- Another way to conceptualize data quality
- Better understanding of model fit and use
- Methods to communicate prior knowledge to the calibration pipeline

...among other useful features. Being aware of these factors can save you a lot of heartache down the road, even if you continue to use a calibration process that only uses accuracy as a metric.



You might have guessed, but Tangram Vision is already addressing this heartache through the Tangram Vision Platform by providing both accuracy and precision for every calibration process we run. Calibration is a core part of our platform, and we hope to offer the best experience available.

WHY CALIBRATION MATTERS FOR AI/ML

The Evolution of Image-Based Inputs for Machine Learning

In the beginning, there were pictures. Simple, static pictures of all sorts of scenes and objects: Dogs. Cars. Houses. People. Fed into systems like DeepMind, they trained classifiers which then generated their own hallucinated outputs from those inputs. Some of the results were predictably hilarious. But some were also incredibly realistic, and hard to discern from the real-world inputs from which they were derived.

Next, came pre-recorded video. In this phase of image-based machine learning (hereafter referred to as "ML"), these video streams were often used to train systems to separate objects from environments, or to discern different environmental states based on multiple observed factors. These formed some of the first data sets to be used to train machine learning systems used for autonomous vehicles (aka, an "AV"). But, and this is important to note, they were not used in real-time during an AV's operation.

Now, however, the applications that rely on imagery-trained ML systems have significantly leveled up. These new applications are low-latency and they *are* real-time. They span a range of industries and use cases, from self driving cars to supply chain robots.

Real-Time Image-Based Machine Learning Relies on Reliable Sensing Inputs

The real-time nature of these applications presents a critical challenge that must be properly managed by the sensor systems that feed them: the data that they receive must be accurate, as it is processed and analyzed in real-time to determine the immediate and subsequent actions of the application.

Sensors, as we know, can fail in multiple ways during deployment:

- Intermittent communication faults (for instance, a flaky USB connection)
- External impediments (for instance, road dust occluding a camera lens)
- Unnoticed or inconsistent shifts in sensor calibration (for instance, a sensor with an insecure mounting)
- Improperly configured sensors (for instance, a camera that does not operate properly in typical daylight settings)

Worse yet, sensors often fail silently. They still emit a data stream, and the ML system assumes that that means the sensors are functioning 100% properly. However, if any of the above scenarios are occurring, they will be sending anomalous data into the ML system. This can create corrupted data sets that lead to immediate errors, or compounding errors that increase over time.

The Impact of Silent Sensor Failures on ML Systems: An Example

Consider the below scenario:

- The System: A new autonomous taxi designed to operate in urban settings

- The Task: Recognize when the autonomous taxi is too close to a curb while in motion

- The Key Parameter: Distances under 1.2m present an unsafe condition due to the presence of bike lanes and cyclists in urban settings

- The Inputs: GPS data; wheel encoders to measure distance traveled; LiDAR sensors and HDR CMOS sensors to capture scene geometry

- The Problem: Silent LiDAR sensor calibration fault due to a loosened sensor mount; extrinsic calibration shifts by 3cm in two axes when AV is in forward motion; returns to near-calibrated state at rest. AV driven during rain results in water occlusion on CMOS sensors.

- The Result: Undetected anomalous data that becomes part of the training set causes the AV to operate within 0.7m of the curb, which is 0.5m too close.

Ensuring Reliable, Accurate Sensor Inputs for ML

Some of the current thinking around sensor failures in machine learning systems suggests that the errors themselves are a valid input for training. This is true, to a degree. The challenge, however, is that the system needs to be able to recognize that a sensor is failing and properly classify that input as such.

This is easier to do under controlled circumstances. In the real world, relying on these processes can become a riskier proposition.

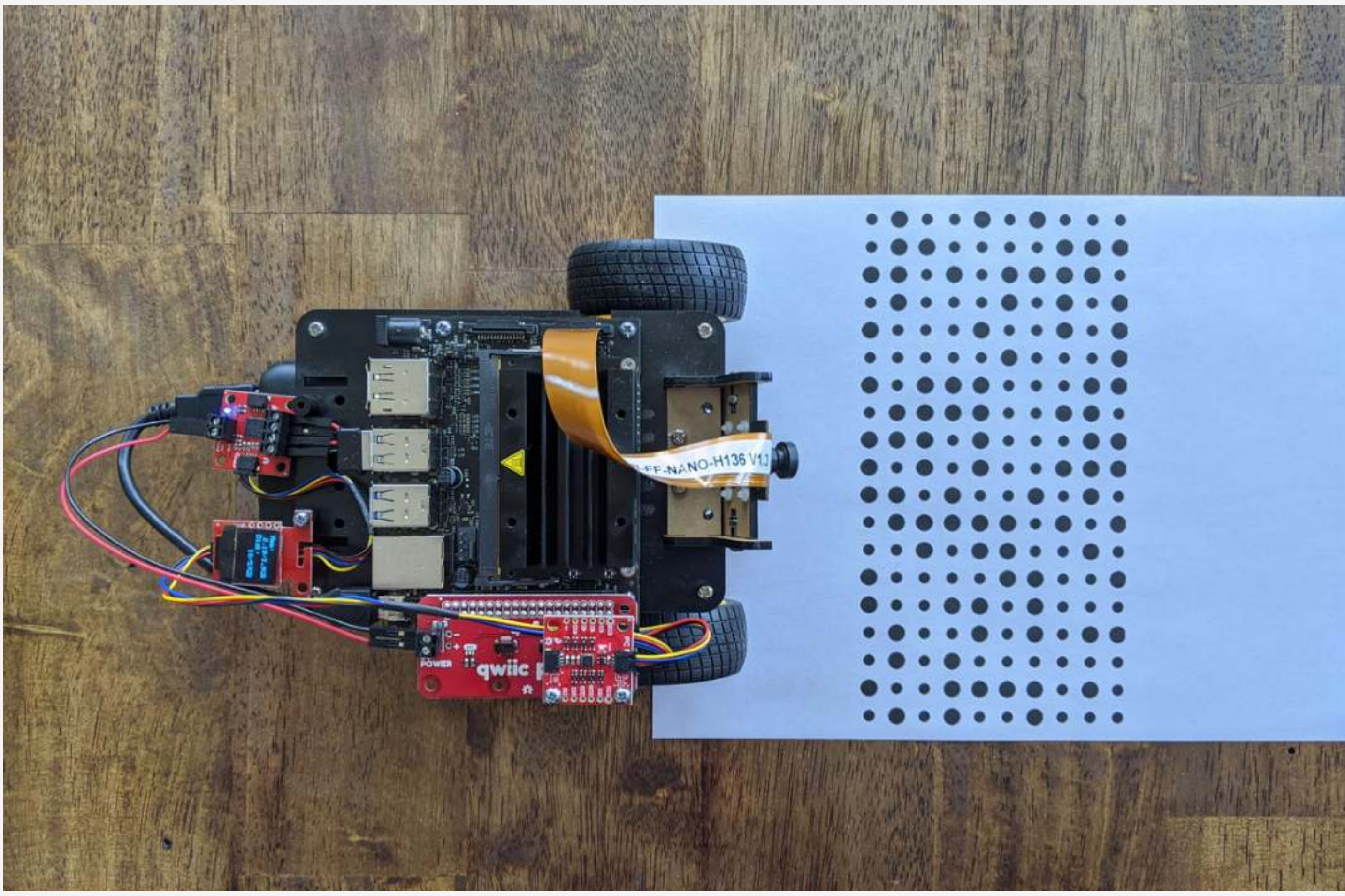
Therefore, the best practice remains ensuring that sensors operate accurately and reliably as much as possible. Achieving this requires effort. It involves:

- Capturing and analyzing statistically significant subsets of sensor logs to identify and properly classify common failure states
- Visually assessing sensors for external impediments after sessions when failures are detected
- Confirming sensor calibration and recalibrating sensors frequently
- Testing sensors with multiple configurations, and then capturing and analyzing sensor logs to identify operational deficiencies before determining the optimal configuration

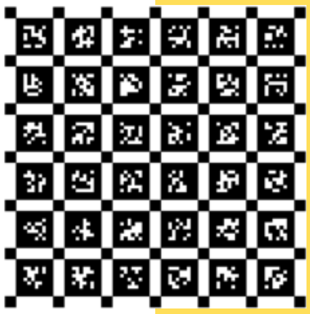
This is not an insignificant amount of work. It requires data analysis, sensor expertise, calibration expertise and copious amounts of engineering time. But, as is the case with any data-driven system, it's a matter of garbage in, garbage out. The investment in proper sensor operation pays off when data sets deliver the expected outcomes, and the ML-powered agent can operate successfully in the real world.

APPENDIX: RESOURCES

75 / Printable Calibration Targets
79 / Calibration Reading List
81 / About Tangram Vision & The Tangram Vision Platform



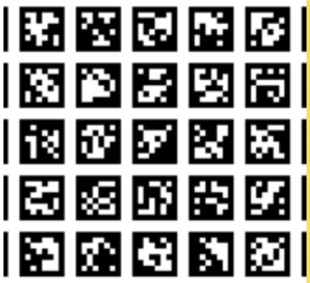
CALIBRATION TARGETS



Kalibr/AprilCal

6X6 GRID

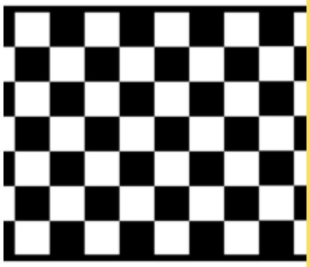
Instructions: Full-size target on page 76. Print edge-to-edge on a standard 8.5 x 11 sheet of paper. Target can be increased in size and printed in larger formats. Mount to as flat and rigid of a surface as possible, such as a mirror, glass pane, or straightened foam core.



AprilCal

5X7 GRID

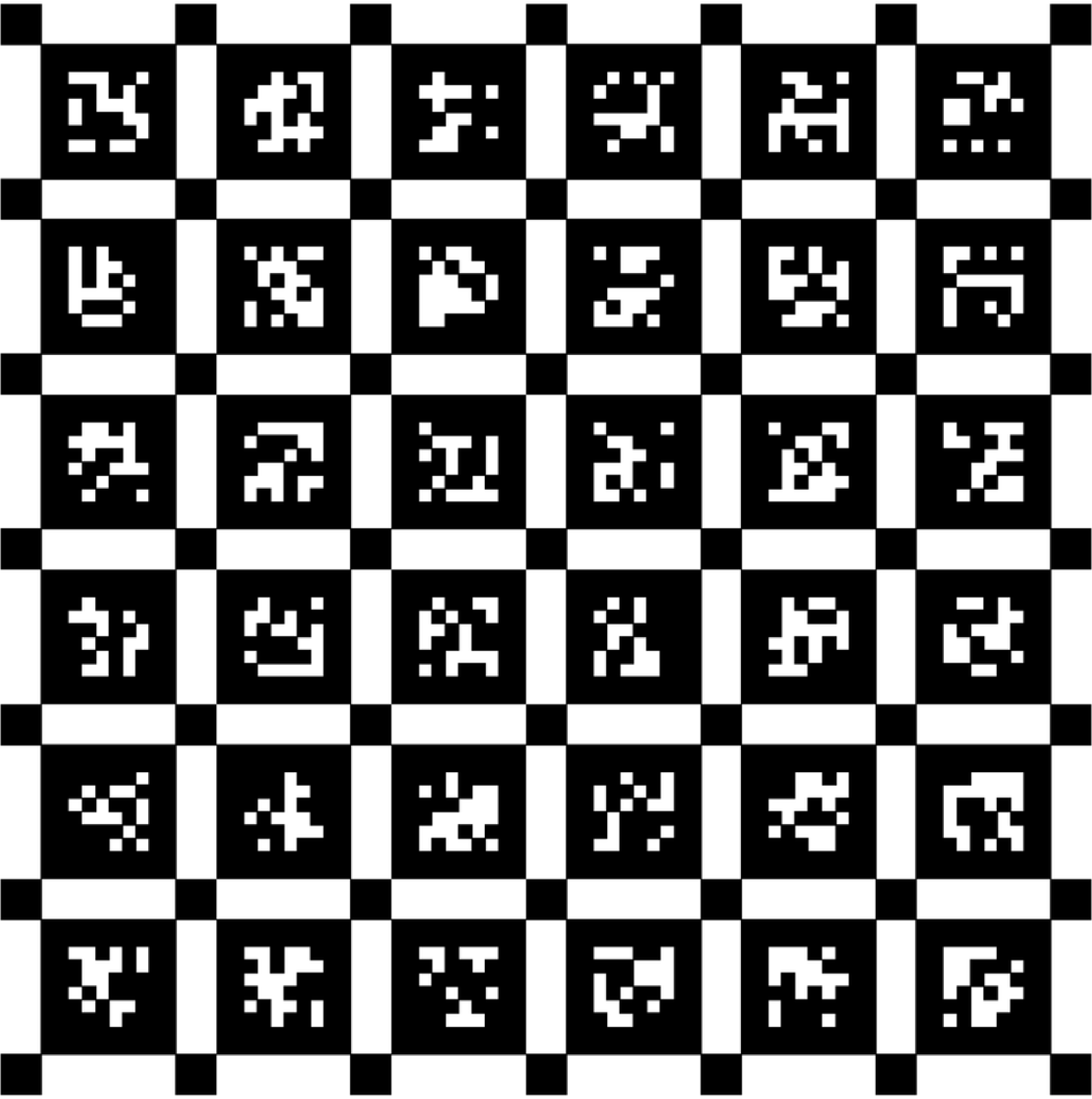
Instructions: Full-size target on page 77. Print edge-to-edge on a standard 8.5 x 11 sheet of paper. Target can be increased in size and printed in larger formats. Mount to as flat and rigid of a surface as possible, such as a mirror, glass pane, or straightened foam core.

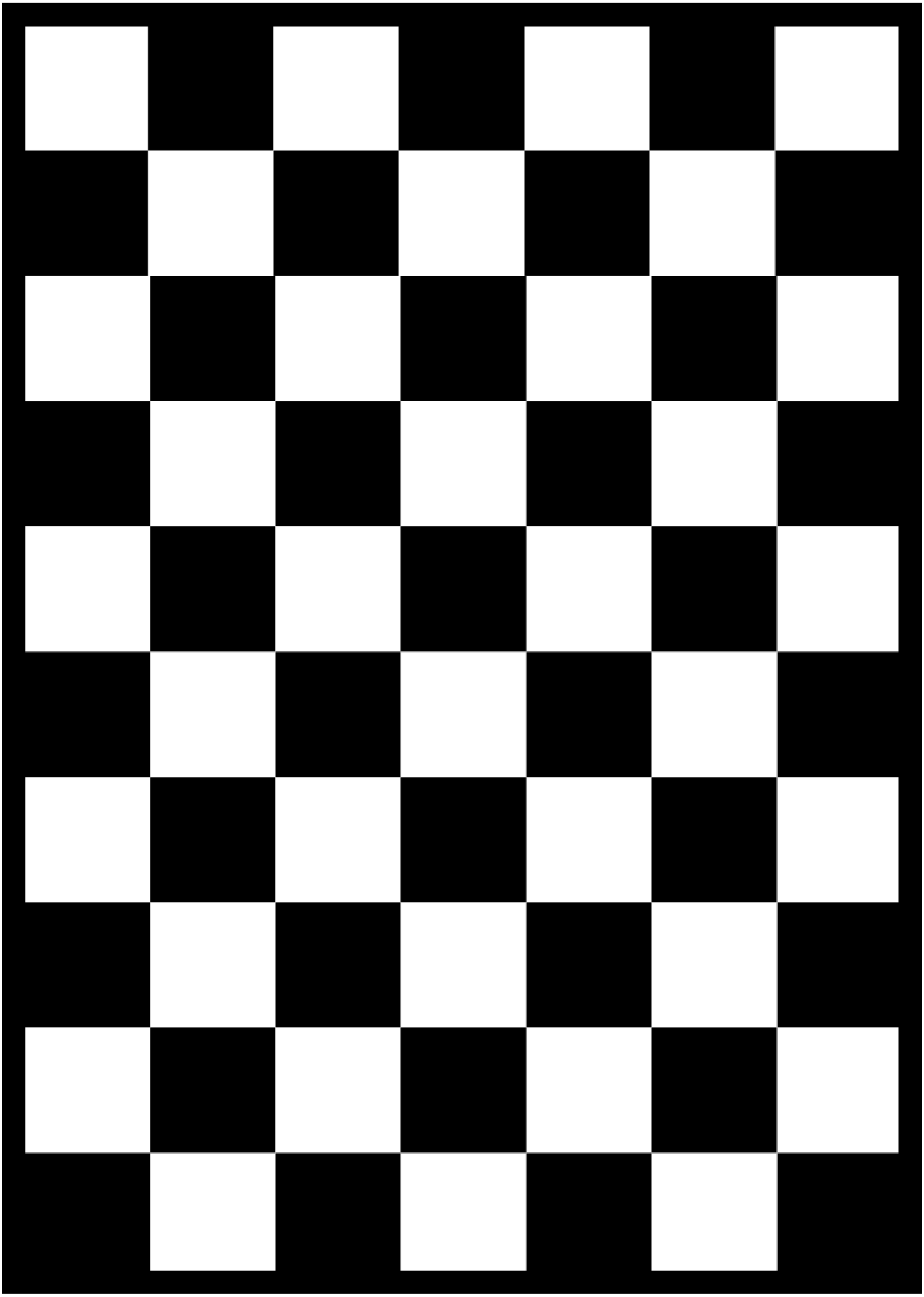


OpenCV

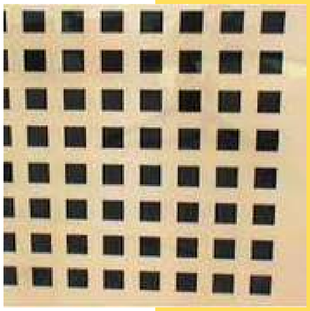
7X6 GRID

Instructions: Full-size target on page 78. Print edge-to-edge on a standard 8.5 x 11 sheet of paper. Target can be increased in size and printed in larger formats. Mount to as flat and rigid of a surface as possible, such as a mirror, glass pane, or straightened foam core.





CALIBRATION READING LIST

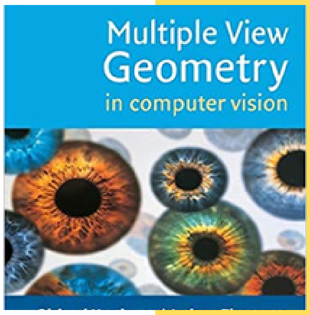


A Flexible New Technique for Camera Calibration

ZHENGYOU ZHANG

One of the seminal works in calibration, and the source for the multi-pose calibration process that we now now as "Zhang's Method." The link in the title above goes to the original paper, including updates to August 2008.

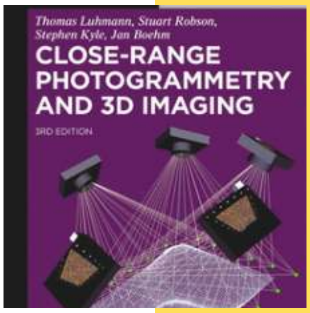
Image credit: Microsoft Research, Microsoft Corporation



Multiple View Geometry In Computer Vision

RICHARD HARTLEY, ANDREW ZISSERMAN

If you've studied computer vision in college, you will have been assigned to read some, or all, of this important textbook. For both beginning and advanced computer vision engineers, *Multiple View Geometry* is a critical text to review to understand the principles underlying much of perception.

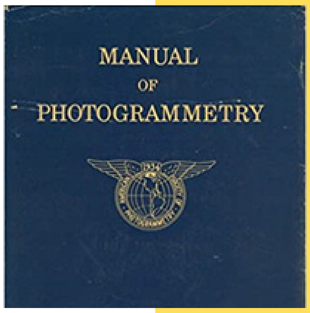

Thomas Luhmann, Stuart Robson, Stephen Kyle, Jan Boehm
CLOSE-RANGE PHOTOGRAMMETRY AND 3D IMAGING
3RD EDITION

Close-Range Photogrammetry and 3D Imaging

THOMAS LUHMANN, STUART ROBSON,
STEPHEN KYLE, JAN BOEHM

An essential primer on the mathematics and methodology behind photogrammetry. Many of the techniques and applications found within this book are directly applicable to common computer vision tasks involving mapping and navigation.

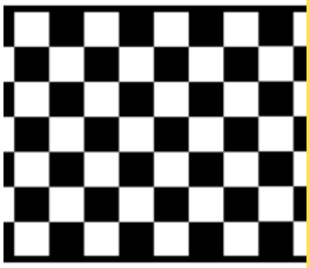
CALIBRATION READING LIST



Manual of Photogrammetry, Sixth Edition

J. CHRIS MCGLONE, EDITOR

Along with "Close-Range Photogrammetry and 3D Imaging", the Manual of Photogrammetry provides a comprehensive reference to the techniques and methods that underpin many mapping, reconstruction, and navigation tasks performed in the field of computer vision.



Decentred Lens Systems

A.E. CONRADY

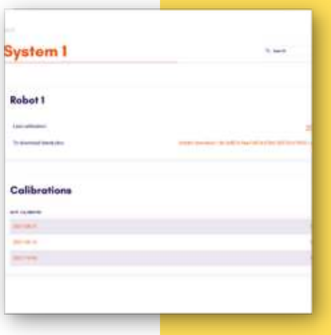
The original paper behind the Conrady-Brown model commonly used for camera calibration. A great way to understand the principles and mathematics behind calibration system inputs.

THE TANGRAM VISION PLATFORM

Sensor Stability with TVMux

RUNTIME, SENSOR FUSION

Integration tools simplify prototyping by making any sensor, any modality plug-and-play. Just connect all your sensors to the Tangram Vision Platform and start streaming instantly. After deployment, integration tools manage sensor stability to provide ongoing uptime in harsh environments.



System 1

Robot 1

Calibrations

Data Management with the Hub

PERCEPTION MANAGEMENT, HEALTH CHECKS

Deploying 100 robots with eight sensors each means managing 800 sensors. A spatial asset database tracks all deployed sensors, including operational health indicators like uptime, errors, calibrations, and more. This data can be accessed via the Tangram Vision Hub or by API.

Sensor Optimization with TVCal & TVFuse

MULTIMODAL CALIBRATION, DATA MANAGEMENT

Eliminate lengthy development cycles by deploying world-class multi-modal calibration and sensor fusion from day one. The Tangram Vision Platform is constantly evolving, with new modalities and manufacturers being added regularly so you won't have to.

To learn more about all of these modules and more, visit www.tangramvision.com.

CONTACT US

www.tangramvision.com
info@tangramvision.com
@tangramvision