Image Transformations in Homogeneous Coordinates

Leo Dorst

March 31, 2016

Abstract

When you do image processing or computer graphics, you naturally have to deal with motions and projective views of moved elements. In standard linear algebra, rotations are easy to treat with orthogonal matrices, but translations are not even linear operations. Therefore standard 3D linear algebra is not a complete tool for the treatment of motions and projections in 3D. However, there is a neat trick: by embedding 3D space in a 4D representational space, the linear algebra of that 4D space is precisely what we need to do projections and motions in the 3D space, and they even become unified as the same kind of operation. This trick is known as *homogeneous coordinates* (we'll find out why). The same trick can be used in one dimension less, to conveniently compute with projective distortions in 2D images by using the linear algebra of a 3D representative space. We treat homogeneous coordinates in that form, and show how they permit you to use standard numerical linear algebra techniques to find transformations.

1 Homogeneous Coordinates in 2D

The translation $T_{\mathbf{t}}$ of a position vector $\mathbf{x}$ over a translation vector $\mathbf{t}$ is simply:

$$\mathbf{x} \;\mapsto\; T_{\mathbf{t}}(\mathbf{x}) \equiv \mathbf{x} + \mathbf{t}.$$

On coordinates, this just has the effect that $(x_1, x_2)^{\mathsf{T}}$ becomes $(x_1 + t_1, x_2 + t_2)^{\mathsf{T}}$. Yet this simple operation is not a linear transformation, for it does not have the linear properties:

$$\begin{aligned} T_{\mathbf{t}}(\alpha\mathbf{x}) &\neq \alpha\, T_{\mathbf{t}}(\mathbf{x}) \\ T_{\mathbf{t}}(\mathbf{x}+\mathbf{y}) &\neq T_{\mathbf{t}}(\mathbf{x}) + T_{\mathbf{t}}(\mathbf{y}). \end{aligned}$$

Verify for yourself that both fail. Because it is not linear, you cannot represent translation as a matrix operating on a vector.

On the other hand, the seemingly more involved operation of rotation over an angle ϕ *is* linear, and you have seen in your linear algebra class that a rotation over ϕ in 2D is representable by the matrix

$$R_\phi = \begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix}.$$

However, this is merely a rotation around the origin. If you would want to rotate around a point at a location $\mathbf{t}$, you would have to combine this with translations. First bring the situation back to the origin by a translation over $-\mathbf{t}$, then rotate, finally put it back at $\mathbf{t}$. That gives the following operation to turn over ϕ around an axis at $\mathbf{t}$:

$$\mathbf{x} \;\mapsto\; R_\phi(\mathbf{x} - \mathbf{t}) + \mathbf{t}.$$

Verify for yourself that this is also not a linear operation, and therefore cannot be represented as a matrix.

A general rigid body motion can always be written as a rotation followed by a translation (check that for the general rotation we just treated!), and the translation part tends to make that transformation non-linear. This is not a major problem, but it does make it awkward to combine successive rigid body motions, or use established linear algebra techniques to estimate motions from data, et cetera. It is worth a bit of effort to re-represent the motions so that they do become linear: they are then representable by matrices, which can be multiplied for successive motions and for which we have least squares methods to perform estimation.[1] It makes rigid body motions tractable by Matlab, or other matrix-based packages.

The trick behind homogeneous coordinates is to embed the vector $\mathbf{x} = (x_1, x_2)^\mathsf{T}$ in one more dimension:

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \mapsto \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}.$$

In that 3D representation you do the math, and when you need to interpret the result you just revert by leaving out the last coordinate (for now; we will make this a bit more general later on). It works, verify for yourself that a translation $T_\mathbf{t}$ is now represented by the matrix:

$$T_\mathbf{t} = \begin{pmatrix} 1 & 0 & t_1 \\ 0 & 1 & t_2 \\ 0 & 0 & 1 \end{pmatrix}.$$

and the rotation by:

$$R_\phi = \begin{pmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Through matrix multiplication, you can compute that a general rigid body motion, represented as a rotation R_ϕ followed by a translation $T_\mathbf{t}$ is now represented by the single matrix:

$$A = \begin{pmatrix} \cos\phi & -\sin\phi & t_1 \\ \sin\phi & \cos\phi & t_2 \\ 0 & 0 & 1 \end{pmatrix}.$$

This immediately pays off when you need to transform a lot of points $\mathbf{p}$, $\mathbf{q}$ et cetera. By the manipulations of linear algebra, you are allowed to just make each of them the column of a data matrix and multiply that by the transform matrix A: the corresponding columns of the result then give the transformed points:

$$\begin{pmatrix} \cos\phi & -\sin\phi & t_1 \\ \sin\phi & \cos\phi & t_2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 & q_1 & \\ p_2 & q_2 & \cdots \\ 1 & 1 & \end{pmatrix}.$$

This is the basis of fast hardware for computer graphics. Note that you always should remember to add a 1 as the last entry of the points to make this work. The point at the origin in 2D is therefore represented as $(0, 0, 1)^\mathsf{T}$, *not* as $(0, 0, 0)^\mathsf{T}$!

[1] Also, the origin is often an arbitrary location in the problem, so why should rotations through origin axes be special? We want a general rotation representation!

After a while, you will be comfortable denoting the matrix A more compactly in terms of its rotation and translation parts as:

$$A = \begin{pmatrix} R_\phi & \mathbf{t} \\ \mathbf{0}^\mathsf{T} & 1 \end{pmatrix}.$$

(where R_ϕ denotes the 2×2 orthogonal matrix of the rotation in 2D, $\mathbf{t}$ the translation vector, and $\mathbf{0}^\mathsf{T} = (0, 0)$). With a bit of practice, you learn to work with those 'block matrices' directly:

$$\begin{pmatrix} R_\phi & \mathbf{t} \\ \mathbf{0}^\mathsf{T} & 1 \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ 1 \end{pmatrix} = \begin{pmatrix} R_\phi \mathbf{x} + \mathbf{t} \\ 1 \end{pmatrix}.$$

It is then also easy to show that 3D representative vectors with a last component of zero do have a sensible meaning: they are the *directions* of 2D space. When you transform them under a rigid body motion A, they only feel the rotation of the motion.

$$\begin{pmatrix} R_\phi & \mathbf{t} \\ \mathbf{0}^\mathsf{T} & 1 \end{pmatrix} \begin{pmatrix} \mathbf{u} \\ 0 \end{pmatrix} = \begin{pmatrix} R_\phi \mathbf{u} \\ 0 \end{pmatrix}.$$

Note that a position remains a position, and a direction remains a direction, under these rigid body motions.

As an insightful aside, this matches well the familiar procedure to specify matrices column-by-column, if you know what they are supposed to do on each of your basis vectors. We have the three basis vectors $(1, 0, 0)^\mathsf{T}$, $(0, 1, 0)^\mathsf{T}$ and $(0, 0, 1)^\mathsf{T}$, which we now interpret as: the x-unit-direction vector, the y-unit-direction vector, and the point at the origin. We know what needs to happen to them: the direction vectors are rotated to $(\cos\phi, \sin\phi, 0)^\mathsf{T}$ and $(-\sin\phi, \cos\phi, 0)^\mathsf{T}$, in a straighforward extension of what the rotations in 2D do to the unit vectors. And the point at the origin should end up at the location $(t_1, t_2)^\mathsf{T}$, so it should become $(t_1, t_2, 1)^\mathsf{T}$. Those are indeed the columns of the rigid body motion matrix.

The matrix for the inverse of a rigid body motion can be computed by a matrix inversion of the matrix A. That is structurally simple, but computationally a bit more expensive than it needs to be. The inverse of a rotation R_ϕ followed by a translation $T_\mathbf{t}$ is of course a translation over $-\mathbf{t}$ followed by a rotation over $-\phi$. That does not give us the matrix, though, for the standard form requires the rotation to be done first. A bit of thought about motions (especially tracking the behavior of the origin point to find the proper translation component) shows how to rewrite:

$$A^{-1} = (T_\mathbf{t} R_\phi)^{-1} = R_{-\phi} T_{-\mathbf{t}} = T_{(-R_{-\phi}\mathbf{t})} R_{-\phi},$$

so that the inverse of A is:

$$\begin{pmatrix} R_\phi & \mathbf{t} \\ \mathbf{0}^\mathsf{T} & 1 \end{pmatrix}^{-1} = \begin{pmatrix} {R_\phi}^\mathsf{T} & -{R_\phi}^\mathsf{T}\mathbf{t} \\ \mathbf{0}^\mathsf{T} & 1 \end{pmatrix}.$$

(Justify for yourself the various usages of ${}^{-1}R_\phi$, $R_{-\phi}$ and ${R_\phi}^\mathsf{T}$ in these formulas!) Verify by direct computation that the above is indeed the correct inverse of a rigid body motion!

The above pattern of treatment for 2D easily generalizes to n-dimensional rigid body motions.

2 Affine Transformations

With the rigid body transformations working so nicely in their matrix form, we can try to extend the principle. A 2D rigid body motion has only 3 parameters (an angle ϕ and a vector $\mathbf{t}$), because

the R-part needs to be an orthogonal transformation.[2] Let us define the more general matrix transformation:

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u \\ v \\ 1 \end{pmatrix}. \tag{1}$$

Here the coefficients a_{ij} forming the matrix A are unrestrained. This is still a rather tidy transformation, for it transforms points into points (the last coefficient remains 1 if it was 1) and directions into directions (the last coefficient remains 0 if it was 0). Therefore it transforms lines into lines (we show this more formally in Section 5 below). Moreover, two parallel directions transform to the same new direction, so *parallel lines remain parallel*.

This transformation is capable of making an arbitrary parallelogram out of a unit square at the origin. In fact, that fully defines it. We can give the ingredients of that square as the point at the origin (which is represented as $(0,0,1)^\mathsf{T}$) and the two sides as direction the vectors $(1,0,0)^\mathsf{T}$ and $(0,1,0)^\mathsf{T}$. You immediately see that the transformation makes out of this a point at the location $(a_{13}, a_{23})^\mathsf{T}$, with the first side now extending by the direction vector $(a_{11}, a_{21})^\mathsf{T}$ (and therefore not necessarily of unit length), and the second side in the direction $(a_{12}, a_{22})^\mathsf{T}$. The other corner point of the square was reached as $(1,1,1)^\mathsf{T} = (0,0,1)^\mathsf{T} + (1,0,0)^\mathsf{T} + (0,1,0)^\mathsf{T}$, and is therefore fully determined by the linearity properties of the transformation: it is at the sum of the transformed direction vectors. So the result is a parallelogram.

Such transformations are called *affine transformations*. You can use them to transform between arbitrary parallelograms. (Because the two parallelograms can both be made out of the unit square, by affine transformations P_1 and P_2, the transformation from parallelogram 1 to parallelogram 2 can be made as $P_2 P_1^{-1}$. Verify that the inverse of an affine transformation is affine, and that the product of two affine transformations is again an affine transformation.) Draw a picture to convince yourself! It should even be possible to write a tool that allows a user to specify 3 points in an original image, and 3 corresponding points in the desired result image, and deduce how all of the other points should be transformed.

Let us develop the math for that. The corresponding point pairs are: $(u_i, v_i)^\mathsf{T}$ goes to $(x_i, y_i)^\mathsf{T}$, for $i = 1, 2, 3$. They give three matrix correspondences:

$$\begin{aligned}
\begin{pmatrix} x_1 \\ y_1 \\ 1 \end{pmatrix} &= \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ v_1 \\ 1 \end{pmatrix} \\
\begin{pmatrix} x_2 \\ y_2 \\ 1 \end{pmatrix} &= \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_2 \\ v_2 \\ 1 \end{pmatrix} \\
\begin{pmatrix} x_3 \\ y_3 \\ 1 \end{pmatrix} &= \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_3 \\ v_3 \\ 1 \end{pmatrix}.
\end{aligned}$$

These are in fact 6 equations for 6 unknowns, so we should be able to solve this. The art is to solve them nicely, using the capabilities of linear algebra, and the Matlab software based on it. Let us take the two equations for the first data point correspondence, and write them out.

$$\begin{aligned}
a_{11}u_1 + a_{12}v_1 + a_{13} &= x_1 \\
a_{21}u_1 + a_{22}v_1 + a_{23} &= y_1
\end{aligned}$$

[2] Show that in 2D, an orthogonal transformation has only 1 parameter. Check that this is true for both rotations and reflections!

The unknowns are the a_{ij}, and the equations are linear in these unknowns, so we should be able to write them as matrix equations in terms of the vector of unknown elements $\mathbf{a} \equiv (a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23})^\mathsf{T}$. And we can! Verify that the above system of 6 equations is equivalent to:

$$\begin{pmatrix} u_1 & v_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & u_1 & v_1 & 1 \\ u_2 & v_2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & u_2 & v_2 & 1 \\ u_3 & v_3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & u_3 & v_3 & 1 \end{pmatrix} \begin{pmatrix} a_{11} \\ a_{12} \\ a_{13} \\ a_{21} \\ a_{22} \\ a_{23} \end{pmatrix} = \begin{pmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ x_3 \\ y_3 \end{pmatrix}. \tag{2}$$

Therefore we get an equation of the form:

$$D\,\mathbf{a} = \mathbf{d},$$

with both the matrix D and the vector $\mathbf{d}$ completely determined by the data that the user specified about the transformation she desired. If the data was sensible (i.e., did not specify a degenerated parallelogram in source or destination image), the matrix D should be invertible. Then we find the coefficients of the affine matrix simply as the entries of:

$$\mathbf{a} = D^{-1}\mathbf{d}.$$

Done. Using the analogy with the code in Section 2.7 of the Lecture Notes on the facet model, it should be simple to convert these equations into a working Matlab program.

If you use this in the context of a desired transformation from image 1 to image 2, you typically need the inverse of the affine transformation (see Section 2.6 on Geometrical Operators, equation 2.5). You can either estimate the transformation A and then invert it, or directly estimate the inverse mapping by switching the roles of (x, y) and (u, v) in the above derivation.

The extension of these 2D principles to affine transformations in n-D should be straightforward. How many parameters will an n-D affine transformation have?

3 Projective Transformations

Of course affine transformations are not the most general you can do with homogeneous coordinates. You could in principle take a general 3×3 matrix and apply that to a homogeneous vector:

$$\lambda \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix} \begin{pmatrix} u \\ v \\ 1 \end{pmatrix}.$$

Now the third coordinate will not remain 1 (for positions) or 0 (for directions).[3] In the above, we have called the third coordinate λ, and pulled it outside the result vector $(x, y, 1)^\mathsf{T}$. We would still like to interpret the result as a point at location (x, y), so we see that we need an extended recipe for the interpretation of a homogeneous vector: a general vector $(x_1, x_2, x_3)^\mathsf{T}$ should be interpreted as denoting a point at the location $(x_1/x_3, x_2/x_3)$.

With that, there is a bit of arbitrariness in the matrix M with the m_{ij} coefficients as well: multiplying them all by a constant generates a different transformation, multiplying the result by that constant; but we interpret what it does to the 2D positions as being exactly the same. So

[3] So we have to be somewhat careful. If $m_{31}u + m_{32}v + m_{33} = 0$, a point at (u, v) has become a direction. You may also have directions (u, v) that become points (under what condition?). Which projective phenomena correspond to these mathematical special cases?

for us, it is the 'same' transformation. This implies that we can somehow normalize the matrix M, for instance by setting $m_{33} = 1$. There are therefore really only 8 parameters in the matrix M that are relevant to our transformation. (The particular choice $m_{33} = 1$ loses generality, for now we do not have the matrix for which m_{33} would be zero. A more stable choice is to normalize such that $\sum_{i,j} m_{ij}^2 = 1$, but you don't really need to normalize it at all to use it.)

The transformation described by this matrix has different names, alternatively called a *projective transformation* (for it describes how a slide transforms when you project it under an angle onto a screen), a *collineation* (since straight lines remain straight lines, we will prove that below), or a *homography* (the same property but now expressed in Greek rather than in Latin). It can transform arbitrary quadrilaterals into arbitrary quadrilaterals. But then we should be able to write a user interface in which the user indicates of four point pairs (u_i, v_i) to which (x_i, y_i) each of them should transform. This is subtly different from the affine problem, and leads to a rather different solution in linear algebra.

First we consider the transformation for a particular data point i. We should remember that the scaling factor λ also depends on the data, so we write that as λ_i, to show that it may be different for each data point i. The full transformation is then:

$$\begin{pmatrix} \lambda_i x_i \\ \lambda_i y_i \\ \lambda_i \end{pmatrix} = M \begin{pmatrix} u_i \\ v_i \\ 1 \end{pmatrix}, \tag{3}$$

where (u, v) is a point in the original image, (x, y) is a point in the projected image and M is a 3×3 projection matrix:

$$M = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix} \tag{4}$$

Writing it out in actual coordinates, you find for the first point correspondence a total of 3 equations:

$$\begin{aligned} m_{11}u_1 + m_{12}v_1 + m_{13} &= \lambda_1 x_1 \\ m_{21}u_1 + m_{22}v_1 + m_{23} &= \lambda_1 y_1 \\ m_{31}u_1 + m_{32}v_1 + m_{33} &= \lambda_1. \end{aligned}$$

But of course we are not interested in λ_1, so we should eliminate it. That leaves 2 equations for this point correspondence:

$$\begin{aligned} m_{11}u_1 + m_{12}v_1 + m_{13} &= m_{31}u_1x_1 + m_{32}v_1x_1 + m_{33}x_1 \\ m_{21}u_1 + m_{22}v_1 + m_{23} &= m_{31}u_1y_1 + m_{32}v_1y_1 + m_{33}y_1 \end{aligned}$$

For the 4 point correspondences we therefore get 8 equations, precisely the right number to solve this problem of determining M uniquely (apart from the arbitrary scaling factor, of course). We are fortunate that these equations are linear in the vector $\mathbf{m}$ of unknown quantities m_{ij}. Write

them into a matrix equation, you should get:

$$\begin{pmatrix} u_1 & v_1 & 1 & 0 & 0 & 0 & -x_1u_1 & -x_1v_1 & -x_1 \\ 0 & 0 & 0 & u_1 & v_1 & 1 & -y_1u_1 & -y_1v_1 & -y_1 \\ u_2 & v_2 & 1 & 0 & 0 & 0 & -x_2u_2 & -x_2v_2 & -x_2 \\ 0 & 0 & 0 & u_2 & v_2 & 1 & -y_2u_2 & -y_2v_2 & -y_2 \\ . & . & . & . & . & . & . & . & . \\ u_n & v_n & 1 & 0 & 0 & 0 & -x_nu_n & -x_nv_n & -x_n \\ 0 & 0 & 0 & u_n & v_n & 1 & -y_nu_n & -y_nv_n & -y_n \end{pmatrix} \begin{pmatrix} m_{11} \\ m_{12} \\ m_{13} \\ m_{21} \\ m_{22} \\ m_{23} \\ m_{31} \\ m_{32} \\ m_{33} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ . \\ 0 \\ 0 \end{pmatrix}, \tag{5}$$

for $n = 4$ (we need the general form later). So in essence, we have to solve the problem:

$$D\mathbf{m} = \mathbf{0},$$

where the matrix D is completely determined by the data of the corresponding point pairs.

Such an equation has the trivial solution $\mathbf{m} = \mathbf{0}$, but we do not want that – it would make the whole projection matrix M equal to zero. Now we use a sensible scaling for M (and therefore, for $\mathbf{m}$). If we use $\|\mathbf{m}\| = 1$ (meaning, the square root of sum of the squares of the m_{ij} equals 1), we can solve the problem exactly using standard linear algebra. We are basically asking for $\mathbf{m}$ to be in the kernel (nullspace) of the matrix D. In the problem we have, D is an 8×9 matrix with general coefficients, so the kernel is 1-dimensional. Therefore the solution is simply $\mathbf{m} \in \ker(D)$ (in Matlab, the command null(D) returns a vector that spans the kernel (or nullspace), so that reads simple m = null(D)).

To get the points of the result image, we should again use M^{-1}, and we can either obtain that by inversion of M, or estimate it directly by switching the roles of input and output points in the correspondences.

4 Stable Estimation of Transformations

The user interfaces just described are not very exact, since the user has to denote points on the screen. You can of course determine the intended transformation more accurately by processing more points. For instance, if you have a Go board in the image, you could have the user specify where more than 3 points (for affine) or more than 4 points (for projective) should end up in the result image.

The basic mapping equations do not change, you just get more of them. In our solution methods, this makes D (and $\mathbf{d}$ for affine) bigger, although the number of unknowns in the transformation of course remains the same. Standard linear algebra still gives the solutions, but they are no longer exact; they are approximations, finding a solution for the unknown vector that minimizes the error norm. In effect, it finds the 'best' matrix in a least square sense on the matrix coefficients.

The affine transformations led to an equation of the form:

$$D\,\mathbf{a} = \mathbf{d}.$$

Originally, D was a general 6×6 matrix, and this has an inverse. With n data point correspondences, D is $2n \times 6$, and the equation can then only be solved approximately (if $n > 3$). The least squares solution to such equations is described in any text on numerical linear algebra, and leads to the pseudo-inverse (or Moore-Penrose inverse). We quickly reconstruct the reasoning leading to the solution.

The only values $D\mathbf{a}$ can take are in the subspace $\mathrm{im}(D)$, i.e. the 'image' of the matrix D. The shortest distance between such a vector and $\mathbf{d}$ is achieved by the value for $\mathbf{a}^\star$ such that $\mathbf{d} - D\mathbf{a}^\star$ is perpendicular to $\mathrm{im}(D)$. It should therefore be perpendicular to all elements that span $\mathrm{im}(D)$, i.e. to all columns of D. That implies that $D^\mathsf{T}(\mathbf{d} - D\mathbf{a}^\star) = 0$. The matrix $D^\mathsf{T}D$ is invertible and that produces the least squares solution

$$\mathbf{a}^\star = \left(D^\mathsf{T}D\right)^{-1} D^\mathsf{T}\mathbf{d}$$

You should verify that in the case of 3 data point correspondences, you get the previous solution involving the inverse.

The projective transformations led to the equation:

$$D\,\mathbf{m} = \mathbf{0},$$

For n data point correspondences, we now have a matrix D of size $2n \times 9$, with $2n$ being at least 8 (remember, there is a scaling freedom in $\mathbf{m}$). This equation cannot be solved exactly (D has no non-trivial null space), so we need a method to find the $\mathbf{m}$ that makes the outcome $D\mathbf{m}$ as small as possible.

The non-trivial 'best' solution $\mathbf{m}^\star$ of the homogeneous equation $D\mathbf{m} = \mathbf{0}$ is found by selecting for $\mathbf{m}^\star$ *the last column of the matrix V that results from the singular value decomposition of the matrix D*. Appendix B of the Lecture notes explains why this is so; geometrically that last column of V is the spatial unit direction that gets reduced most by D. Since this solution corresponds to the smallest singular value of D, the magnitude of that is a measure of its accuracy (its deviation from the exact zero result that was required). You may verify that this method correctly computes the kernel for the 8×9 matrix of the original problem. What happens for a 6×9 matrix based on 3 point correspondences?

5 Transforming hyperplanes

In homogeneous coordinates, points (elements of dimension 0) are easily represented as vectors, and easily transformed. But hyperplanes are not much harder. Hyperplanes are flat elements of dimension $(n-1)$ in n-dimensional space: planes in 3D, lines in $2D$. Let us focus on lines in 2D in our explanation.

A point with coordinates (x, y) lies on a line if there is a relationship between x and y of the form:

$$y = ax + b.$$

This denotes a line with slope a which intercepts the y-axis at b. This is actually not a very good way of writing the line, for we cannot even represent the y-axis itself. It is more symmetrical to say that there is the following line-like relationship between x and y:

$$l_1 x + l_2 y + l_3 = 0.$$

The conversion to the earlier form is easy, for $a = -l_1/l_2$ and $b = -l_3/l_2$, but now the y-axis can be represented (Q: How?). Of course any multiple of the l_i would work just as well, since it gives essentially the same equation.

Now we introduce homogeneous coordinates for the point at (x, y), so set $\mathbf{x} \equiv (x, y, 1)^\mathsf{T}$, and put the l_i in a 3-component vector $\mathbf{l} = (l_1, l_2, l_3)^\mathsf{T}$ as well. Then we can write the equation of the line as $\mathbf{l} \cdot \mathbf{x} = 0$, or in terms of matrices as:

$$\mathbf{l}^\mathsf{T}\mathbf{x} = 0.$$

This has a zero on the right, and that makes it a *homogeneous equation* to mathematicians. (That is the origin of the term homogeneous coordinates: they turn these kinds of hyperplane equations into homogeneous equations.)

Now suppose that we transform the image using a projective transformation. We know that in homogeneous coordinates this can be represented by a matrix M, acting on any point $\mathbf{x}$ to make a new point $\mathbf{y} = M\mathbf{x}$. What happens if we transform all the points that were originally on the line? They all satisfy $0 = \mathbf{l}^\mathsf{T}\mathbf{x}$, and substituting $\mathbf{x} = M^{-1}\mathbf{y}$ we find that the corresponding points $\mathbf{y}$ must satisfy:

$$0 = \mathbf{l}^\mathsf{T}M^{-1}\mathbf{y} = ((M^{-1})^\mathsf{T}\mathbf{l})^\mathsf{T}\mathbf{y}.$$

But that is again the equation of a line, now characterized by the vector $(M^{-1})^\mathsf{T}\mathbf{l}$. Therefore the projective transformation of a line is again a line (and moreover, we can specify precisely which line because we have found the transformation of its parameter vector $(l_1, l_2, l_3)^\mathsf{T}$: it is the transpose of the inverse of M). You can check that they even transform intersecting lines into intersecting lines[4], and that explains why they are sometimes called *collineations*.

Since affine transformations are a special case of projective transformations, they also turn lines into lines (even parallel lines into parallel lines), and so do rigid body motions (moreover, they preserve the angle between two lines).

[4] If you agree to say that parallel lines intersect at infinity.