EECS 442 Homework 3 Lecture Notes

Foreword

This file reviews the fitting of transformations from 2D points to 2D points. Some of this is from lecture; some of it also appears in in textbooks. We think you would benefit from a summarized version in the same conventions as the lecture slides rather than having to translate conventions.

The Setup

Recall from lecture the setup:

1. Suppose we have a set of k correspondences $(x_i, y_i) \leftrightarrow (x'_i, y'_i)$, or the pixel (x_i, y_i) in image 1 corresponds with the pixel at (x'_i, y'_i) in image 2.

2. It can be useful to avoid notational clutter to also define $\mathbf{p}_i^T \equiv [x_i, y_i, 1]$ and $\mathbf{p}_j'^T \equiv [x'_j, y'_j, 1]$. These are the homogeneous coordinates for the given pixels. When doing derivations, you should never to assume that the last coordinate of $\mathbf{p}_i$ is 1; however, once you build a specific matrix for a problem, the "scale" of the coordinate is a free parameter, and so you can always set it to one.

3. *Recall:* Homogeneous coordinates are the 2D coordinates you know and love with an extra dimension. So they have three coordinates $[a, b, c]$. When you have a homogeneous coordinate $[a, b, c]$, you can convert into an ordinary coordinate $[a/c, b/c]$. If $c = 0$, this corresponds to what's called "the line at infinity" (where parallel lines converge); for stitching you should have no points with $c = 0$. Two homogeneous coordinates $\mathbf{p}$ and $\mathbf{p}'$ represent the same point in the image if they are proportional to each other, or $\mathbf{p} = \lambda\mathbf{p}'$ for $\lambda \neq 0$.

4. We'd like to find a *transformation* or model between the images. This transformation is parameterized by a small number of parameters. It is important to note that the precise direction (is it the transformation from image 1 to image 2 or is it the reverse?) will be the subject of many bugs and conventions used by libraries you call. There is no shame in inverting it if you think things look wrong. We may often refer to these transformations (or their parameters) in different shapes if this is notationally convenient. For instance, a homography can be expressed as a 3x3 matrix of the form:

$$\mathbf{H} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \begin{bmatrix} \mathbf{h}_1^T \\ \mathbf{h}_2^T \\ \mathbf{h}_3^T \end{bmatrix} \quad \text{where} \quad \mathbf{h}_1 = \begin{bmatrix} a \\ b \\ c \end{bmatrix}, \ \mathbf{h}_2 = \begin{bmatrix} d \\ e \\ f \end{bmatrix}, \ \mathbf{h}_3 = \begin{bmatrix} g \\ h \\ i \end{bmatrix}. \tag{1}$$

Here, we have simply pulled the rows out. While you might want to use the 3×3 matrix $\mathbf{H}$ to transform points (i.e., $\mathbf{p}'_i \equiv \mathbf{H}\mathbf{p}_i$ if things fit right), you might also think of the homography as being described by vertically stacking $\mathbf{h}_1, \mathbf{h}_2, \mathbf{h}_3$ together to make a 9×1 vector which you could call $\mathbf{h}$ (the vector version of $\mathbf{H}$). This isn't a convenient form factor for doing matrix multiplications, but if you want to formulate an optimization problem $\arg\min \|\mathbf{A}\mathbf{h}\|_2^2$, it's great. But in the end, the data is the same.

Similarly, an affine transformation can be expressed as either a 3×3 matrix, or a 2×3 matrix $\mathbf{M}$ of the form:

$$\mathbf{M} = \begin{bmatrix} m_1 & m_2 & t_x \\ m_3 & m_4 & t_y \\ 0 & 0 & 1 \end{bmatrix} \text{ or } \begin{bmatrix} m_1 & m_2 & t_x \\ m_3 & m_4 & t_y \end{bmatrix}, \tag{2}$$

and we may want to gather all of the terms into one column vector $[m_1, m_2, m_3, m_4, t_x, t_y]^T \in \mathbb{R}^6$ or in two matrices

$$\mathbf{S} = \begin{bmatrix} m_1 & m_2 \\ m_3 & m_4 \end{bmatrix} \quad \mathbf{t} = \begin{bmatrix} t_x \\ t_y \end{bmatrix}. \tag{3}$$

In the end it's all the same transformation.

Reading a homography

So you have a homography $\mathbf{H}$ that your system produced. How do you read it? It's tricky but you can get a sense looking at individual terms while imagining the rest are identity.

First, **always** normalize by dividing by the *last* coordinate if you want to read it. You can arbitrarily scale the homography without changing the operation it performs. If we use the notation from Eqn. 1, you will always have to divide by the transformed point $\mathbf{H}[x, y, 1]^T$ by $gx + hy + i$. It's better if you don't have to do this every time you want to read a component. If you have a zero in i, then things have really gone wrong. This should not happen in the data we have given you (or rather, no homography with a zero in i should have the best agreement with the correspondences you find).

Suppose you now have the matrix, but normalized. Keep in mind that you are doing this operation:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \equiv \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{4}$$

It can be helpful to examine what each set of terms does if the rest of the matrix is set to the identity matrix (i.e., not transforming the points). This is when $b, c, d, f, g, h = 0$ and $a, e = 1$.

1. **Linear transform (a, b, d, e):** In general, this block is inscrutable without further processing. This could be a rotation matrix if $[a, b]$ and $[d, e]$ are orthonormal (i.e., unit length and orthogonal to each other); it could be a rotation followed by a shearing; it could be a scaling and shearing. In the end, all matrices are a composition of a rotation, non-isotropic scaling, and rotation. But interpreting these does not tell you what's going on in an interpretable way. Precisely reading this bit is not worth your trouble – the only thing to make sure is that the orders of magnitude of the matrix (and sign) are roughly right.

2. **Translation factors (c, f):** This block always represents translation in x, y (c, f). These should be about the size of the displacement between the two images (measured in one of the image's pixels).

3. **Scaling factors (a, e):** These factors scale the values of x and y. You should not have very large values here unless you are transforming between two very differently sized images. If these are negative, your image is being flipped.

4. **Shear factors (b, d):** Each of these factors applies a shear mapping in the absence of other terms. This does not mean, however, that if $b, d \neq 0$, then there is a shear mapping – if the transform is a rotation in 2D, these will be non-zero as well (but a, e will also not be 1).

5. **Affine transformation (a, b, c, d, e, f):** If g, h are 0, then the rest of the homography is just an affine mapping. Remember that affine transformations preserve lines and parallelism.

6. **Perspective factors (g, h):** These control how much "perspective effect" (to give a loose sense) is incorporated. If $g, h = 0$, then parallel lines *remain* parallel; setting $g, h \neq 0$ is precisely what allows this bend. If you represent the values of x, y in pixels (usually measured in hundreds), these should be small. If they are large, something has gone wrong (or you somehow have $\mathbf{H}^T$ so that the translation factors are in the perspective factors' location).

Fitting Models to Data

Here are some useful mathematical facts about least-squares and transformations, expressed concisely and put in context of what they're good for in computer vision:

1. **Least-squares:** Suppose I have a matrix $\mathbf{A} \in \mathbb{R}^{N \times M}$ and vector $\mathbf{b} \in \mathbb{R}^N$. The vector $\mathbf{x}^* \in \mathbb{R}^M$ that minimizes $||\mathbf{Ax} - \mathbf{b}||_2^2$ is given by $(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b}$, commonly referred to as the least-squares solution. In practice, you should call `np.linalg.lstsq`, which handles the numerical catches to this well.

   What does this do for us? If I pick my variables $\mathbf{A}, \mathbf{b}$ carefully, I can make the expression $||\mathbf{Ax} - \mathbf{b}||_2^2$ be something meaningful with regards to $\mathbf{x}$ (which typically contains the parameters of the model). In particular, in our problems, we want to construct a matrix $\mathbf{A}$ and vector $\mathbf{b}$ so that they depend only on the *data* we have (and not the models). Often in transformation-fitting problems, you put terms involving the image coordinates from image 1 in $\mathbf{A}$; then, if you put the terms of the transformation into $\mathbf{x}$, $\mathbf{Ax}$ contains the model's assumptions about where the points go. Then if $\mathbf{b}$ are just the points in image 2, $||\mathbf{Ax} - \mathbf{b}||_2^2$ is the sum of squared distances and measures how closely the transform maps image 1 to image 2 given our correspondences. Then, when we solve for the optimal value for $\mathbf{x}$, we get the optimal terms of the model.

2. **Homogeneous Least-squares:** This is often good enough, but some setups are hard to do in standard least-squares. Many computer vision problems are defined *up to a scale*. I can multiply the homography matrix by a non-zero constant, and this has no impact on what the transformation actually does. One of the standard tricks for dealing with this is called *homogeneous least squares*.

   Suppose I have a matrix $\mathbf{A} \in \mathbb{R}^{N \times M}$. The unit vector $\mathbf{x}^* \in \mathbb{R}^M$ s.t. $||\mathbf{x}^*||_2^2 = 1$ that minimizes $||\mathbf{Ax}||_2^2$ is given by the eigenvector of $(\mathbf{A}^T\mathbf{A})$ that has the smallest eigenvalue. In some sense, you can also read the above expression like minimizing $||\mathbf{Ax} - \mathbf{0}||_2^2$ (just like setting $\mathbf{b} = \mathbf{0}$ in the above least-squares setup).

   What does this do for our lives? Again, like setting up a least-squares problem, you put values in $\mathbf{A}$ carefully so that if $\mathbf{x}^*$ contains the transformation parameters (i.e., is the vertical stacking of $\mathbf{h}_1, \mathbf{h}_2, \mathbf{h}_3$), $\mathbf{Ax}^*$ should be close to zero if the transformation/model has fit well.

3. **Degrees of freedom:** Counting degrees of freedom can be tricky. We'll do three here. The general strategy is to try to identify how many parameters there are that you are free to pick. One way to do this is to add one for every parameter, and then subtract off for every parameter that you are not allowed to pick and then subtract one for every dimension that's invariant (e.g., subtract one if the object is known up to a scale). This can be dangerous since you can double count constraints – one might imply another. Another option is to pick the parameters one by one, but only picking them if they're unconstrained. Then subtract for invariances (like scale).

Homographies: you have 9 parameters $(+9)$; the matrix is known only up to a scale (-1). So it's 8.

Homogeneous coordinates of the form $[u, v, w]$ *with* $w \neq 0$*:* you have three coordinates $(+3)$; the value is known only up to a scale (-1). So it's 2.

Rotation matrices: This is a 3×3 matrix $(+9)$ with the following constraints: the columns and rows are each unit length, the rows and columns are orthogonal, and the determinant is 1 (not -1). A lot of these constraints imply the others. An easier way to go about this is to pick the parameters column-by-column. (Column 1) You can pick two out of the three coordinates in the first column $(+2)$ since the last coordinate is determined by the fact that the column must have unit norm. (Column 2) You can pick one coordinate in the second column $(+1)$ since the other two are constrained by the fact that that column also has unit norm and must be orthogonal to the first column. (Column 3) The third column is determined entirely by the first two: it is orthogonal to both (reducing the space you can pick to a line), is unit norm (reducing it two two unit norm vectors on that line), and makes the determinant positive (picking the specific vector).

4. **Number of samples needed to constrain the problem:** Given an object whose parameters we want to estimate (e.g., a homography), we often fit the parameters of the object to data. Generally each data point gives n equations which *must* be true if the model is functioning properly; the object will generally have a number of free parameters/degrees of freedom which we'll call p (e.g., for homographies $p = 8$). Up to a small caveat, suppose we have k data points, if $nk \leq p$, then the system is *underconstrained* – the data points' equations specify a family of models rather than a specific one; if $nk = p$, then the system is *constrained* – the data points' equations precisely specify a model; if $nk \geq p$ then the system is *overconstrained* – almost certainly no model satisfies all the data points' equations and so we must find a best-fit model (e.g., via least-squares).

   Caveat: In practice, one has to be careful with any of these setups: this analysis assumes that given k points, you get nk equations. This isn't necessarily true – suppose all the points were the same, then you have only n independent equations. In computer vision, this is usually referred to as *general position* and corresponds to the way that most points will behave (as opposed to something like trying to fit a 3D plane to three collinear points).

Fitting Affine Transformations

We'll next re-present the setups from class. In general, we will have k correspondences. Each correspondence will produce multiple rows of the matrices $\mathbf{A}$ (and if appropriate $\mathbf{b}$). You are free to allocate these rows as zeros and fill them in, or write them out and concatenate them. If your solution doesn't work, you may want to re-enter the matrices in a *different way*.

$$\begin{bmatrix} \vdots \\ x'_i \\ y'_i \\ \vdots \end{bmatrix} = \begin{bmatrix} & & \vdots & & & \\ x_i & y_i & 0 & 0 & 1 & 0 \\ 0 & 0 & x_i & y_i & 0 & 1 \\ & & \vdots & & & \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ t_x \\ t_y \end{bmatrix} \quad \text{or} \quad \mathbf{b} = \mathbf{A}\mathbf{x} \tag{5}$$

where $\mathbf{b} \in \mathbb{R}^{2k \times 1}$, $\mathbf{A} \in \mathbb{R}^{2k \times 6}$, $\mathbf{x} \in \mathbb{R}^{6 \times 1}$. This setup may not seem intuitive at first – it's designed for least-squares and not for humans. But if you multiply $\mathbf{Ax}$ out, you should get $m_1 x_i + m_2 y_i + t_x$ in the top row and $m_3 x_i + m_4 y_i + t_y$ in the bottom row.

So: given that $\mathbf{Ax}$ is where the model says the point should be, $||\mathbf{b}-\mathbf{Ax}||_2^2$ is computing the sum-of-squared differences between the points, or:

$$\begin{aligned} ||\mathbf{b}-\mathbf{Ax}||_2^2 &= \sum_{1\le i\le k}(x'_i-(m_1x_i+m_2y_i+t_x))^2+(y'_i-(m_3x_i+m_4y_i+t_y))^2 \\ &= \sum_{1\le i\le k}||[x'_i,y'_i]-[m_1x_i+m_2y_i+t_x,m_3x_i+m_4y_i+t_y]||_2^2 \end{aligned} \quad (6)$$

Finding the $\mathbf{x}$ that minimizes things is thus the best-fit model.

Fitting Homographies

This setup is a trickier and depends on homogeneous coordinates. The key to the setup is to try to make an equation which is true when $\mathbf{p}'_i \equiv \mathbf{H}\mathbf{p}_i$ (recall that $\mathbf{p}_i \equiv [x_i, y_i, 1]$ from earlier).

The problem is that we test homogeneous equivalence ($\mathbf{x} \equiv \mathbf{y}$), which is true if and only if the two vectors are proportional (there is a $\lambda \neq 0$ such that $\mathbf{x} = \lambda\mathbf{y}$). Testing or whether two vectors are proportional by finding the λ is ugly. So we use the following trick: the cross product $\mathbf{x}\times\mathbf{y} = 0$ if $\mathbf{x}$ and $\mathbf{y}$ are proportional. This trick is non-intuitive for many but arises from the fact that the magnitude of the cross product is equal to the area of the parallelogram between the two vectors. If they both face the same way, that parallelogram's zero!

Once we have this, what we'll do is calculate what $\mathbf{H}\mathbf{p}_i$ looks like. If we have $\mathbf{H}$ broken into its rows, this doesn't look so annoying. Note that that $\mathbf{h}_1$ is the column vector containing a, b, and c from the homography. We lay it on its side to get it into the matrix properly. If we calculate things out, we get:

$$\mathbf{H}\mathbf{p}_i = \begin{bmatrix} \mathbf{h}_1^T \\ \mathbf{h}_2^T \\ \mathbf{h}_3^T \end{bmatrix} \mathbf{p}_i = \begin{bmatrix} \mathbf{h}_1^T\mathbf{p}_i \\ \mathbf{h}_2^T\mathbf{p}_i \\ \mathbf{h}_3^T\mathbf{p}_i \end{bmatrix}. \quad (7)$$

If the points are correctly related by the homography, then both of the following statements are true.

$$\mathbf{p}'_i \equiv \mathbf{H}\mathbf{p}_i \equiv \begin{bmatrix} \mathbf{h}_1^T\mathbf{p}_i \\ \mathbf{h}_2^T\mathbf{p}_i \\ \mathbf{h}_3^T\mathbf{p}_i \end{bmatrix} \quad \text{or, equivalently,} \quad \mathbf{p}'_i \times \begin{bmatrix} \mathbf{h}_1^T\mathbf{p}_i \\ \mathbf{h}_2^T\mathbf{p}_i \\ \mathbf{h}_3^T\mathbf{p}_i \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \mathbf{0} \quad (8)$$

This gives a set of three equations involving $\mathbf{h} = [\mathbf{h}_1, \mathbf{h}_2, \mathbf{h}_3]$ via the following steps: explicitly calculate each term of the cross-product and set it to 0; shuffle things around so that the expression is a linear expression with respect to all elements of $\mathbf{h}$ (with some of the terms equal to 0). It's algebraic manipulation and not particularly enlightening. The only catch is that not all three equations you get are linearly independent, so you end up with two equations per correspondence.

In the end, for every correspondence you have, you get two rows of an ugly matrix in terms of image 1 points $\mathbf{p}_i = [x_i, y_i, 1]^T$ and image 2 points (x'_i, y'_i):

$$\begin{bmatrix} & \vdots & \\ \mathbf{0}^T & -\mathbf{p}_i^T & y'_i\mathbf{p}_i^T \\ \mathbf{p}_i^T & \mathbf{0}^T & -x'_i\mathbf{p}_i^T \\ & \vdots & \end{bmatrix} \begin{bmatrix} \mathbf{h}_1 \\ \mathbf{h}_2 \\ \mathbf{h}_3 \end{bmatrix} = \begin{bmatrix} \vdots \\ 0 \\ 0 \\ \vdots \end{bmatrix} \quad \text{or} \quad \mathbf{Ah} = \mathbf{0} \quad (9)$$

where $\mathbf{0}^T = [0,0,0]$ (i.e., 1×3) and $\mathbf{p}_i^T = [x_i, y_i, 1]$ (i.e., also 1×3). For sizes, each row of this matrix is 1×9. If $\mathbf{h}$ is the 9×1 vector of parameters, the first row of the expression says that $[\mathbf{0}^T, -\mathbf{p}_i^T, y_i\mathbf{p}_i^T]\mathbf{h} = 0$, or $\mathbf{0}^T\mathbf{h}_1 - \mathbf{p}_i^T\mathbf{h}_2 + y'_i\mathbf{p}_i^T\mathbf{h}_3 = 0$.

So: Given that $\mathbf{Ah}$ is zero when the transformation exactly maps points together, $\mathbf{h}$ is invariant to its scale, it seems sensible to use the homogeneous least-squares. In short: you construct $\mathbf{A}$ by putting the terms in the right place, then compute the eigenvector of $\mathbf{A}^T\mathbf{A}$ that corresponds to the smallest eigenvalue.

Caveat: The only caveat is that this expression is convenient to minimize but not geometrically meaningful; in practice, though, it's usually good enough. That said, professional systems usually do a few steps of standard nonlinear optimization afterwards directly minimizing the distance between $\mathbf{Hp}_i$ and $\mathbf{p}'_i$ (treated as 2D points by dividing out by the last coordinate). There is, for what it's worth, an even more incorrect least-squares minimization that weights the per-point error arbitrarily.

Putting Stitching Together

We can put the whole pipeline together now.

1. *Find and Describe Features:* First, you detect image features like corners and extract descriptors near these locations in both images.

   You will call some code that will give you what's called a *keypoint*. The keypoint consists of a *location* $\mathbf{p}_i$ (e.g., found by Harris Corner detection or the Laplacian of Gaussians) as well as a *descriptor* $\mathbf{d}_i$ (e.g., histogrammed gradients) that describes the image content near that region in a fixed-length vector (e.g., 128 dimensions). The keypoint often is used to describe the combination of the two, and whether you are referring to its location or descriptor is implied from context. There's typically also some preferred distance that's used to measure how similar the two descriptors are. In this homework, we've pre-processed the data so that the square of the L_2 distance works.

   This gives you a set of locations and descriptors $\{\mathbf{p}_i, \mathbf{d}_i\}_i$ in image 1 and a similar set $\{\mathbf{p}'_j, \mathbf{d}'_j\}_j$ in image 2. In general if keypoint i in image 1 has a good match in image 2, it is the nearest neighbor to $\mathbf{d}_i$ in $\{\mathbf{d}'_j\}_j$, or $j^* = \arg\min_j \|\mathbf{d}_i - \mathbf{d}'_j\|_2^2$.

2. *Match Features Between Images:* We can *always* find a nearest neighbor for a descriptor, but there's no guarantee it's at all any good. Thresholding on $\|\mathbf{d}_i - \mathbf{d}'_j\|_2^2$ usually doesn't work: distances in high-dimensional spaces are often of dubious value. The solution is this clever trick: find the nearest neighbor j^* and second nearest neighbor j^{**}. Then compute

   $$r = \frac{\|\mathbf{d}_i - \mathbf{d}_{j^*}\|_2^2}{\|\mathbf{d}_i - \mathbf{d}_{j^{**}}\|_2^2}, \tag{10}$$

   and if this distance ratio is small (i.e, the nearest neighbor is substantially closer than the next-nearest neighbor), then the match is good. Otherwise, you can safely throw it away. The intuition is that this accepts a match if it's distinctive.

   By doing this trick, you can create a collection of matches *between* the images. This gives matches for a *subset* of the original keypoints. Note: *There are absolutely no guarantees this matching is one-to-one/bijective. In practice it usually is not.*

3. *Robustly Fit Transformation:* Given the matches between the keypoints' descriptors, you then get a collection of correspondences $\mathbf{p}_i \leftrightarrow \mathbf{p}'_j$. If you have N matches, you then can create a $N \times 4$ matrix of all the locations. You can then safely ignore the original keypoints and just focus on the corresponding locations. By running RANSAC plus a homography fitting routine, you get a transformation between the correspondences.

4. *Warp Image Onto Another:* This is best explained by doing. But once you have the transformation that describes how the matches go from one image to another, you can warp (i.e., transfer and rearrange) all of the pixels. You have to pick one image that doesn't get warped; the other image gets warped to that image's coordinate system. You will almost certainly have to *expand the image* to cover both images. This produces a composite image containing both images.