Computer Vision: Homework 3

CV2022@CSU

1 Instructions

Python Environment We are using Python for this homework. You can find references for the Python standard library here: https://docs.python.org. To make your life easier, we recommend you to install Anaconda for Python (https://www.anaconda.com/download/). This is a Python package manager that includes most of the modules you need. We will make use of the following package: Numpy, Matplotlib, OpenCV.

2 The Setup and Math You May Want

2.1 Homogeneous Coordinates

If you have a 2D point $[u, v]$, you can convert it to a homogeneous coordinate by concatenating 1 and making $[u, v, 1]$. If you have a homogeneous coordinate $[a, b, c]$ you can convert it to an ordinary 2D point by dividing by the last coordinate c, or $[a/c, b/c]$. If $c = 0$, this doesn't work; a point with $c = 0$ is at "the point at infinity" (where parallel lines converge).

Intuition: When we have a point on the imaging plane, we've made it a homogeneous coordinate by concatenating a 1. You can think of this point as really representing a line from the origin to the point on the image plane. We often first define the point to be on a plane with depth 1 although all vectors that face the same direction are equivalent (except when the depth is 0).

Checking for equivalence: Two homogeneous coordinates $\mathbf{p}$ and $\mathbf{p}'$ represent the same point in the image if they are proportional to each other, or there is some $\lambda \neq 0$ such that $\mathbf{p} = \lambda \mathbf{p}'$. Testing if there exists a λ is tricky. Instead, it's useful to generate constraints in computer vision by testing if $\mathbf{p} \times \mathbf{p}' = 0$ (i.e., the cross product is zero). Note the equals! This works because the magnitude of the cross product is determined by the area of a parallelogram with $\mathbf{p}$ and $\mathbf{p}'$ as its sides. If they point in the same direction, this area is zero.

2.1.1 Homogeneous Coordinates and Lines

One of the nice things about homogeneous coordinates is that they make points and lines the same size. If I have a 2D points in homogeneous coordinates $\mathbf{x} \in \mathbb{R}^3$ and $\mathbf{y} \in \mathbb{R}^3$ and a line $\mathbf{l} = [a, b, c]^T \in \mathbb{R}^3$, then $\mathbf{x}$ lies on the line 1 (i.e., $ax + by + c = 0$) if and only if $\mathbf{l}^T\mathbf{x} = \mathbf{x}^T\mathbf{l} = 0$. But critically, you can interpret any 3D vector as a point or a line. This leads to lots of genuinely wonderful results, but also makes dealing with points and lines a lot easier than most of what you're inclined to do based on high school math.

Intuition for Lines: You can think of a line on the image plane as really being a plane that passes through the origin as well as the image plane. The line coordinates are actually the normal of that plane. The usual 3D plane equation involving a normal $\mathbf{n}$ and offset o, namely $\mathbf{n}^T[x, y, z] + o = 0$ is simplified by the fact that the plane passes through the origin so $o = 0$.

Intersection of two lines: given two lines $\mathbf{l}_1$ and $\mathbf{l}_2$, their intersection $\mathbf{p}$ is given by $\mathbf{l}_1 \times \mathbf{l}_2$. This can be seen algebraically by the fact that $\mathbf{p} = \mathbf{l}_1 \times \mathbf{l}_2$ has to be orthogonal to both (i.e., $\mathbf{p}^T\mathbf{l}_1 = 0$ and $\mathbf{p}^T\mathbf{l}_2 = 0$). So naturally, p has to satisfy both lines' equations. Geometrically, you could think of this as constructing a direction that is perpendicular to both planes' normals: this has to lie on both the planes, and therefore is the intersection of the planes.

Line through two points: given two points $\mathbf{x}$ and $\mathbf{y}$, the line through them is given by $\mathbf{x} \times \mathbf{y}$. Algebraically, this follows for the same reason why the intersection of two lines is the cross product. Geometrically, you can think of this as constructing a plane that is perpendicular to both directions. Draw a line from the origin to both points. The origin and both points form a plane whose normal is proportional to $\mathbf{x} \times \mathbf{y}$.

2.2 2D Transformation

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 $\mathbf{p}_j'^T \equiv \left[x'_j, y'_j, 1\right]$. 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. We'd like to find a transformation or model between the images. This transformation is parameterized by a few 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 2D projective transformation (*homography*) can be expressed as a 3×3 matrix of the form:

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

   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, a 2D affine transformation can be expressed as either a 3×3 matrix, or a 2×3 matrix $\mathbf{H}_A$ of the form:

$$\mathbf{H}_A = \begin{bmatrix} m_1 & m_2 & t_x \\ m_3 & m_4 & t_y \\ 0 & 0 & 1 \end{bmatrix} \text{ or } \begin{bmatrix} m_1 & m_2 & t_x \\ m_3 & m_4 & t_y \end{bmatrix}$$

   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

$$\mathbf{A} = \begin{bmatrix} m_1 & m_2 \\ m_3 & m_4 \end{bmatrix} \quad \mathbf{t} = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

   In the end it's all the same transformation.

2.3 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 Eq. (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:

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

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. 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).

2.4 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 $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2$ is given by $\left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T\mathbf{b}$, commonly referred to as the least-squares solution. In practice, you can 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 $\|\mathbf{A}\mathbf{x} - \mathbf{b}\||_2^2$ be something meaningful with regards to 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 $\mathbf{x}$, $\mathbf{A}\mathbf{x}$ contains the model's assumptions about where the points go. Then if $\mathbf{b}$ are just the points in image 2, $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2$ 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 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 $\|\mathbf{A}\mathbf{x}\|_2^2$ is given by the eigenvector of $\left(\mathbf{A}^T\mathbf{A}\right)$ that has the smallest eigenvalue. In some sense, you can also read the above expression like minimizing $\|\mathbf{A}\mathbf{x} - \mathbf{0}\|_2^2$ (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 $\mathbf{h}_1, \mathbf{h}_2, \mathbf{h}_3$) , $\mathbf{A}\mathbf{x}^*$ 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-bycolumn. (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).

2.5 Fitting Affine Transformations

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.

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

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 Ax out, you should get $m_1x_i + m_2y_i + t_x$ in the top row and $m_3x_i + m_4y_i + t_y$ 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:

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

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

2.6 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:

$$\mathbf{H}\mathbf{p}_i = \begin{bmatrix} \mathbf{h}_1^T \\ \mathbf{h}_2^T \\ \mathbf{h}_3^T \end{bmatrix} \mathbf{p}_i = \begin{bmatrix} \mathbf{h}_1^T\mathbf{p}_i \\ \mathbf{h}_2^T\mathbf{p}_i \\ \mathbf{h}_3^T\mathbf{p}_i \end{bmatrix}$$

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

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

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') :

$$\left[\begin{array}{ccc} & \vdots & \\ \mathbf{0}^T & -\mathbf{p}_i^T & y_i' \mathbf{p}_i^T \\ \mathbf{p}_i^T & \mathbf{0}^T & -x_i' \mathbf{p}_i^T \\ & \vdots & \end{array}\right]\left[\begin{array}{c} \mathbf{h}_1 \\ \mathbf{h}_2 \\ \mathbf{h}_3 \end{array}\right]=\left[\begin{array}{c} \vdots \\ 0 \\ 0 \\ \vdots \end{array}\right] \quad \text{or} \quad \mathbf{A h}=\mathbf{0}$$

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 $\left[\mathbf{0}^T,-\mathbf{p}_i^T, y_i \mathbf{p}_i^T\right] \mathbf{h}=0$, 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 $\mathbf{A h}$ 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 $\mathbf{H p}_i$ and $\mathbf{p}_i'$ (treated as 2 D 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.

2.7 Stitching Pipeline

(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.

(2) 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 $j^*=\arg\min_j \left\|\mathbf{d}_i-\mathbf{d}_j'\right\|_2^2$.

(3) 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 $\left\|\mathbf{d}_i-\mathbf{d}_j'\right\|_2^2$ 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

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

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.

(4) 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.

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.

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.

3 Fitting models

Task 1: RANSAC

Supposing we are fitting a 3D plane (i.e., $ax + by + cz + d = 0$). A 3D plane can be defined by 3 points in general position (2 give a line). Plane fitting happens when people analyze point clouds to reconstruct scenes from laser scans. Each question depends on on the previous one. If you are not sure about one previous answer, give an equation in terms of a set of variables. In keeping with other notation that you may find elsewhere, we will refer to the model that is fit in the inner loop of RANSAC as the putative model. Write in your report the minimum number of 3D points needed to sample in an iteration to compute a putative model.

Task 2: Fitting Linear Transforms

Throughout, suppose we have a set of 2D correspondences ($[x'_i, y'_i] \leftrightarrow [x_i, y_i]$) for $1 \le i \le N$.

- Suppose we are fitting a linear transformation, which can be parameterized by a matrix $\mathbf{M} \in \mathbb{R}^{2\times 2}$ (i.e., $[x', y']^T = \mathbf{M}[x, y]^T$).

  Write in your report: the number of degrees of freedom $\mathbf{M}$ has and the minimum number of 2D correspondences that are required to fully constrain or estimate $\mathbf{M}$.

- Suppose we want to fit $[x'_i, y'_i]^T = \mathbf{M}\,[x_i, y_i]^T$. We would like you formulate the fitting problem in the form of a least-squares problem of the form

  $$\arg\min_{m \in \mathbb{R}^4} \|\mathbf{A}\mathbf{m} - \mathbf{b}\|_2^2$$

  where $\mathbf{m} \in \mathbb{R}^4$ contains all the parameters of $\mathbf{M}$, $\mathbf{A}$ depends on the points $[x_i, y_i]$ and $\mathbf{b}$ depends on the points $[x'_i, y'_i]$.

  Write the form of $\mathbf{A}$, $\mathbf{m}$, and $\mathbf{b}$ in your report.

Task 3: Fitting Affine Transforms

Throughout, again suppose we have a set of 2D correspondences $[x'_i, y'_i] \leftrightarrow [x_i, y_i]$ for $1 \le i \le N$. Files: We give an actual set of points in `task3/points_case_1.npy` and `task3/points_case_2.npy`: each row of the matrix contains the data $[x_i, y_i, x'_i, y'_i]$ representing the correspondence. You do not need to turn in your code but you may want to write some file `task3.py` that loads and plots data.

- Fit a transformation of the form

  $$[x', y']^T = \mathbf{S}[x, y]^T + \mathbf{t}, \quad \mathbf{S} \in \mathbb{R}^{2\times 2}, \mathbf{t} \in \mathbb{R}^{2\times 1}$$

  by setting up a problem of the form

  $$\arg\min_{\mathbf{v}\in\mathbb{R}^6} \|\mathbf{A}\mathbf{v} - \mathbf{b}\|_2^2$$

  and solving it via least-squares. Report $(\mathbf{S}, \mathbf{t})$ in your report for `points_case_1.npy`.

  Hint: There is no trick question. Write a small amount of code that does this by loading a matrix, shuffling the data around, and then calling `np.Iinalg.Istsq`.

- Make as scatterplot of the points $[x_i, y_i]$, $[x'_i, y'_i]$ and $\mathbf{S}[x_i, y_i]^T + \mathbf{t}$ in one figure with different colors. Do this for both `points_case_1.npy` and `point_case_2.npy`. In other words, there should be two plots, each of which contains three sets of N points. Save the figures and put them in your report.

  Hint: Look at `plt.scatter` and `plt.savefig`. For drawing the scatterplot, you can do `plt.scatter(xy[:,0], xy[:,1], 1)`. The last argument controls the size of the dot and you may want this to be small so you can set the pattern. As you ask it to scatterplot more plots, they accumulate on the current figure. End the figure by `plt.close()`.

- Write in the report your answer to: how well does an affine transform describe the relationship between $[x, y] \leftrightarrow [x', y']$ for `points_case_1.npy` and `points_case_2.npy`? You should describe this in two or three sentences.

  Hint: what properties are preserved by each transformation?

Task 4: Fitting Homographies

Files: We have generated 9 cases of correspondences for this task. These are named `points_case_k.npy` for $1 \leq k \leq 9$. All are the same format as the previous task and are matrices where each row contains $[x_i, y_i, x'_i, y'_i]$. Eight are transformed letters M. The last case (case 9) is copied from task 3. You can use these examples to verify your implementation of `fit_homography`.

- Fill in `fit_homography` in `homography.py`.

  This should fit a homography mapping between the two given points. Remembering that $\mathbf{p}_i \equiv [x_i, y_i, 1]$ and $\mathbf{p}'_i \equiv [x'_i, y'_i, 1]$, your goal is to fit a homography $\mathbf{H} \in \mathbb{R}^3$ that satisfies

  $$\mathbf{p}'_i \equiv \mathbf{H}\mathbf{p}_i.$$

  Most sets of correspondences are not exactly described by a homography, so your goal is to fit a homography using an optimization problem of the form

  $$\underset{\|\mathbf{h}\|_2^2=1}{\arg\min} \|\mathbf{A}\mathbf{h}\|, \quad \mathbf{h} \in \mathbb{R}^9, \mathbf{A} \in \mathbb{R}^{2N \times 9},$$

  where $\mathbf{h}$ has all the parameters of $\mathbf{H}$.

- Report $\mathbf{H}$ for cases `points_case_1.npy` and `points_case_4.npy`. You must normalize the last entry to 1.

- Visualize the original points $[x_i, y_i]$, target points $[x'_i, y'_i]$ and points after applying a homography transform $T(H, [x_i, y_i])$ in one figure. Please do this for `points_case_5.npy` and `points_case_9.npy`. Thus there should be two plots, each of which contains 3 sets of N points. Save the figure and put it in the report.

4 Image Warping and Homographies

Task 5: Synthetic Views

We asked San Zhang what he's reading, and so he sent us a few pictures. They're a bit distorted since he wants you to get used to `cv2.warpPerspective` before you use it in the next task. He says "it's all the same, right, homographies can map between planes and book covers are planes, no?".

Files: We provide data in `task5/`, with one folder per book. Each folder has: (a) `book.jpg` - an image of the book taken from an angle; (b) `corners.npy` - a numpy containing a 4×2 matrix where each row is $[x_i, y_i]$ representing the corners of the book stored in (top-left, top-right, bottom-right, bottom-left) order; (c) `size.npy` which gives the size of the book cover in inches in a 2D array [height, width].

- Fill in `make_synthetic_view(scenelmage, corners, size)` in `task5.py`.

  This should return the image of the cover viewed head-on (i.e., with cover parallel to the image plane) where one inch on the book corresponds to 100 pixels.

  Walkthrough: First fit the homography between the book as seen in the image and book cover. In the new image, the top-left corner will be at $[x, y] = [0, 0]$ and the bottom-right corner will be at $[x, y] = [100w - 1, 100h - 1]$. Figure out where the other corners should go. Then read the documentation for `cv2.warpperspective`.

- Put a copy of both book covers in your report.

- (Optional) One of these images doesn't have perfectly straight lines. Write in your report why you think the lines might be slightly crooked despite the book cover being roughly a plane. You should write about 3 sentences.

- (Suggestion/optional) Before you proceed, see if you can make another function that does the operation in the reverse: it should map the corners of `synthetic` cover to `sceneImage` assuming the same relationship between the corners of synthetic and the listed corners in the scene. In other words, if you were to doodle on the cover of one of the books, and send it back into the scene, it should look as if it's viewed from an angle. Pixels that do not have a corresponding source should be set to 0. What happens if synthetic contains only ones?

Task 6: Stitching Stuff Together (Optional)

Recall from the introduction that a keypoint has a location $\mathbf{p}_i$ and descriptor $\mathbf{d}_i$. There are many types of keypoints used. Traditionally this course has used SIFT and SURF, but these are subject to patents and installed in only a few versions of OpenCV. Traditionally, this has led to this homework being an exercise in figuring out how to install a very special version of OpenCV (and then figuring out some undocumented features).

We provide another descriptor called AKAZE (plus some other features) in `common.py`. In addition to this descriptor, you are encouraged to look at `common.py` to see if there are things you want to use while working on the homework. The calling convention is: `keypoints, descriptors = common.get_AKAZE(image)` which will give you a $N \times 4$ matrix keypoints and a $N \times F$ matrix descriptors containing descriptors for each keypoint. The first two columns of keypoints contain x, y; the last two are the angle and (roughly) the scale at which they were found in case those are of interest. The descriptor has also been post-processed into something where $\|\mathbf{d}_i - \mathbf{d}'_j\|_2^2$ is meaningful.

Files: We provide you with a number of panoramas in `task6/` that you can choose to merge together. To enable you to run your code automatically on multiple panoramas without manually editing filenames (see also `os.Iistdir`), we provide them in a set of folders.

Each folder contains two images: (a) `p1.jpg`; and (b) `p2.jpg`. Some also contain images (e.g., `p3.jpg`) which may or may not work. You should be able to match all the provided panoramas; you should be able to stitch all except for `florence3` and `florence3_alt`.

- Fill in `compute_distance` in `task6.py`. This should compute the pairwise squared ℓ_2 distance between two matrices of descriptors.
- Fill in `find_matches` in `task6.py`. This should use `compute_distance` plus the ratio test from the foreword to return the matches. You will have to pick a threshold for the ratio test. Something between 0.7 and 1 is reasonable, but you should experiment with it (look output of the `draw_matches` once you complete it). Beware! The numbers for the ratio shown in the lecture slides apply to SIFT; the descriptor here is different so the ratio threshold you should use is different. Hints: look at `np.argsort` as well as `np.take_along_axis`.
- Fill in `draw_matches` in `task6.py`. This should put the images on top of each other and draw lines between the matches. You can use this to debug things. Hints: Use `cv2.line`.
- Put a picture of the matches between two image pairs of your choice in your report.
- Fill in `RANSAC_fit_homography` in `homography.py`. This should RANSACify `fit_homography`. You should keep track of the best set of inliers you have seen in the RANSAC loop. Once the loop is done, please re-fit the model to these inliers. In other words, if you are told to run N iterations of RANSAC, you should fit a homography N times on the minimum number of points needed; this should be followed by a single fitting of a homography on many more points (the inliers for the best of the N models). You will need to set epsilon's default value: 0.1 pixels is too small; 100 pixels is too big. You will need to play with this to get the later parts to work. Hints: when sampling correspondences, draw without replacement; if you do it with replacement you may pick the same point repeatedly and then try to (effectively) fit a model to three points.
- Fill in `make_warped` in `task6.py`. This should take two images as an argument and do the whole pipeline described above. The resulting image should use `cv2.warpperspective` to make a merged image where both images fit in. This merged image should have: (a) image 1's pixel data if only image 1 is present at that location; (b) image 2's pixel data if only image 2 is present at that location; (c) the average of image 1's data and image 2's data if both are present.

Walkthrough:

- There is an information bottleneck in estimating $\mathbf{H}$. If $\mathbf{H}$ is correct, then you're set; if it's wrong, there's nothing you can do. First make sure your code estimates $\mathbf{H}$ right.

- Pick which image you're going to merge to; without loss of generality, pick image 1. Figure out how to make a merged image that's big enough to hold both image 1 and transformed image 2. Think of this as finding the smallest enclosing rectangle of both images. The upper left corner of this rectangle (i.e., pixel $[0, 0]$) may not be at the same location as in image 1. You will almost certainly need to hand-make a homography that translates image 1 to its location in the merged image. For doing this calculations, use the fact that the image content will be bounded by the image corners. Looking at the min, max of these gives you what you need to create the panorama.

- Warp both images to the merged image. You can figure out where the images go by warping images containing ones to the merged images instead of the image and filling the image with os where the image doesn't go. These masks also tell you how to create the average.

Debugging hints:

- Make a fake pair of images by taking an image, rolling it by $(10, 30)$ and then saving it. Debugging this is far easier if you know what the answer should be.

- If you want to debug the warping, you can also provide two images that are crops of the same image, (e.g., I$[100:400, 100:400]$ and I$[150:450, 75:375]$) where you know the homography (since it is just a translation).

- Put merges from two of your favorite pairs in the report. You can either choose an image we provide you or use a pair of images you take yourself.

- Put these merges as `mypanorama1.jpg` and `mypanorama2.jpg` in your zip submission.

- If you would like to submit a panorama, please put your favorite as `myfavoritepanorama.jpg`.

5 Augmented Reality on a Budget (Optional)

If you can warp images together, you can replace things in your reality. Imagine that we have a template image and this template appears in the world but viewed at an angle. You can fit a homography mapping between the template and the scene. Once you have a homography, you can transfer anything. This enables you to improve things.

Task 7: Augmented Reality on a Budget (Optional)

Files: We give a few examples of templates and scenes in `task7/scenes/`. Each folder contains: `template.png`: a viewed-from-head-on/distortion-free/fronto-parallel version of the texture; and `scene.jpg`: an image where the texture appears at some location and viewed at some angle. We provide a set of seals (e.g., the UM seal) that you may want to put on things in `task7/seals`. You can substitute whatever you like.

- Fill in the function `improve_image(scene, template, transfer)` in `task7.py` that aligns `template` to `scene` using a homography, just as in task 6. Then, instead of warping `template` to the image, warp `transfer`. If you want to copy over your functions from task 6, you can either import them or just copy them.

  Hints:

  - The matches that you get are definitely not one-to-one. You'll probably get better results if you match from the template to the scene (i.e., for each template keypoint, find the best match in scene). Be careful about ordering though if you transfer your code!
  - The image to transfer might not be the same size as the template. You can either resize transfer to be the same size as template or automatically generate a homography.

- Do something fun with this. Submit a synthetically done warp of something interesting. If you do something particularly neat to get the system to work, please write this in the report.

Submit in your zip file the following files:

- `myscene.jpg` - the scene
- `mytemplate.png` OR `mytemplate.jpg` - the template. Submit either png or jpg but not both.
- `mytransfer.jpg` - the thing you transfer
- `myimproved.jpg` - your result

Guidelines: If you try this on your own images, here are some suggestions:

- Above all, please be respectful.
- This sort of trick works best with something that has lots of texture across the entire template. The lacroix carton works very well. The `aep.jpg` image that you saw in dithering does not work so well since it has little texture for the little building at the bottom.
- This trick is most impressive if you do this for something seen at a very different angle. You may be able to extend how far you can match by pre-generating synthetic warps of the template (i.e, generate $\mathrm{synth}_i = \mathrm{apply}\left(\mathbf{H}_i, T\right)$ for a series of $\mathbf{H}_i$, then see if you can find a good warping $\hat{\mathbf{H}}$ from synth $_i$ to the scene. Then the final homography is $\hat{\mathbf{H}}\mathbf{H}_i$.

6 Submission

将所有的代码、实验图像和实验报告打包成zip文件(班级-学号-姓名-HW3.zip)通过学校可视化教学平台提交，代码中所有文件路径请使用相对路径(切勿使用绝对路径)