EECS 442 Computer Vision: (Optional) Homework 6

Instructions

- This homework **was** due **due at 11:59:59 p.m. on April 20, 2021**.
- The submission includes two parts:
  1. **To Canvas:** submit a `zip` file of all of your code.
     We have indicated questions where you have to do something in code in red.
     We have indicated questions where we will definitely use an autograder in purple.
     Please be especially careful on the autograded assignments to follow the instructions. Don't swap the order of arguments and do not return extra values.
     If we're talking about autograding a filename, we will be pulling out these files with a script. Please be careful about the name.
     Your zip file should contain a single directory which has the same name as your uniqname. If I (David, uniqname `fouhey`) were submitting my code, the zip file should contain a single folder `fouhey/` containing all required files.
     What should I submit? At the end of the homework, there is a canvas submission checklist provided. We provide a script that validates the submission format here. If we don't ask you for it, you don't need to submit it; while you should clean up the directory, don't panic about having an extra file or two.
  2. **To Gradescope:** submit a `pdf` file as your write-up, including your answers to all the questions and key choices you made.
     We have indicated questions where you have to do something in the report in blue.
     You might like to combine several files to make a submission. Here is an example online link for combining multiple PDF files: https://combinepdf.com/.
     The write-up must be an electronic version. **No handwriting, including plotting questions.** LaTeX is recommended but not mandatory.

Python Environment

We are using Python 3.7 for this course. You can find references for the Python standard library here: https://docs.python.org/3.7/library/index.html. To make your life easier, we **recommend** you to install the latest Anaconda for Python 3.7 (https://www.anaconda.com/download/). This is a Python package manager that includes most of the modules you need for this course.

We will make use of the following packages extensively in this course:

- Numpy (https://docs.scipy.org/doc/numpy-dev/user/quickstart.html).
- Matplotlib (http://matplotlib.org/users/pyplot_tutorial.html).
- OpenCV (https://opencv.org/).

1 Foreword

1.1 The Setup and Math You May Want

This setup is best read alongside the lecture slides.

You already know some of this stuff, but often it takes seeing something a few times to get it.

Preemptively, I am being a bit hand-wavy in parts and I am dancing around using cleaner explanations that depend on more advanced linear algebra. My apologies to the math department.

1.1.1 Cross products

Despite the fact that we effectively live in a 3D space, linear algebra classes often do not teach the cross product $\times$. The cross product of two vectors $\mathbf{a}$ and $\mathbf{b}$, denoted $\mathbf{a} \times \mathbf{b}$, just makes a third vector $\mathbf{c}$ that points in a direction that is orthogonal to $\mathbf{a}$ and $\mathbf{b}$ (i.e., $\mathbf{c}^T\mathbf{x} = 0$ and $\mathbf{c}^T\mathbf{y} = 0$), and has norm $||\mathbf{c}||_2$ that is equal to the area of parallelogram with $\mathbf{a}$ and $\mathbf{b}$ as sides. The orientation of $\mathbf{c}$ is determined by the right-hand rule.

Rather than call `np.cross` or something, you can also compute the cross product with a fixed vector $\mathbf{x} = [x_1, x_2, x_3]$ by the following form involving a skew-symmetric matrix $[\mathbf{x}]_\times$:

$$\mathbf{x} \times \mathbf{y} = [\mathbf{x}]_\times \mathbf{y} = \begin{bmatrix} 0 & -x_3 & x_2 \\ x_3 & 0 & -x_1 \\ -x_2 & x_1 & 0 \end{bmatrix} \mathbf{y} \tag{1}$$

Once you have the cross-product, you can also produce the *triple product* $\mathbf{z}^T(\mathbf{x} \times \mathbf{y})$ or the dot product of $\mathbf{z}$ with a vector that is perpendicular to $\mathbf{x}$ and $\mathbf{y}$. This somewhat cryptic expression is useful in computer vision since it is a good test for co-planarity. Namely, if the triple product is zero, then $\mathbf{z}$ is co-planar with $\mathbf{x}$ and $\mathbf{y}$ (it has zero projection onto the normal for the plane defined by the origin, $\mathbf{x}$, and $\mathbf{y}$).

1.1.2 Homogeneous Coordinates

At this point, you're familiar with homogeneous coordinates, but we'll just recap them.

Homogeneous coordinates are the 2D coordinates that you know and love with an extra dimension. 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 line 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. It's ok if you didn't see this in your Linear Algebra class.

1.1.3 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 $\mathbf{l}$ (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, $\mathbf{p}$ 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 $\mathbf{x} \times \mathbf{y}$.

1.1.4 Camera Projection

Suppose you have a 2D point $\mathbf{p} = [u, v, 1]$ and a 3D point $\mathbf{X} = [x, y, z, 1]$ (capitalized to be consistent with the slides, where it's capitalized to make it more obvious it's 3D). The camera that projects things to the world is composed of three key components: an *intrinsics* matrix $\mathbf{K} \in \mathbb{R}^{3\times3}$, and a rotation $\mathbf{R} \in \mathbb{R}^{3\times3}$ and translation $\mathbf{t} \in \mathbb{R}^3$. The rotation and translation join together to form the *extrinsics* matrix $[\mathbf{R}, \mathbf{t}]$, which is the horizontal concatenation of the two matrices (i.e., a member of $\mathbb{R}^{3\times4}$). The intrinsics matrix $\mathbf{K}$ is intrinsic because it doesn't change with the location of the camera. The extrinsics matrix is the matrix of the camera's properties with respect to the external world.

The matrices aren't all freely choosable. The only one that is freely choosable is the translation, which can be picked arbitrarily. Since $\mathbf{R}$ is a rotation, it only has 3 degrees of freedom despite having 9 coordinates.

In its most general form, $\mathbf{K}$ might just be an upper triangular matrix. But usually $\mathbf{K}$ is assumed to be of the form:

$$\mathbf{K} = \begin{bmatrix} f & 0 & c_u \\ 0 & f & c_v \\ 0 & 0 & 1 \end{bmatrix}. \tag{2}$$

The particular 3-parameter form of $\mathbf{K}$ assumes that pixels are square and there is no skew (i.e., the axes of the image are perpendicular). Both seem like obvious requirements. While they aren't always met in practice, they usually hold (or rather hardware manufacturers spend a lot of energy making sure they hold). Personally, I've only encountered one camera that has non-square pixels – a solar telescope (Hinode/SOT-SP) that creates images by sweeping a line of pixels across the Sun. However, I am told that it's often useful to have parameters that absorb modeling errors (i.e., even if the pixels are square, pretending they might be slightly wrong might handle other unmodeled issues).

In total, projection should satisfy $\mathbf{p} \equiv \mathbf{K}[\mathbf{R}, \mathbf{t}]\mathbf{X}$. This can be grouped in a number of equivalent ways

to get some insight into what's going on. If you multiply all the camera matrices, you can get a 3×4 matrix $\mathbf{M} = \mathbf{K}[\mathbf{R}, \mathbf{t}]$. Then $\mathbf{p} \equiv \mathbf{MX}$. Alternatively, you can think of this as $\mathbf{p} \equiv \mathbf{K}([\mathbf{R}, \mathbf{t}]\mathbf{X})$, or $\mathbf{X}$ gets transformed by $\mathbf{R}$ and $\mathbf{t}$, and then projected by $\mathbf{K}$.

1.1.5 Homogeneous Least-Squares Redux

Often we set up fitting problems where we would like to find p parameters of a model $\mathbf{x}$ (i.e., $\mathbf{x} \in \mathbb{R}^p$) and we can create an equation $\mathbf{a}$ such that if $\mathbf{a}^T\mathbf{x} = 0$, then the model fits well. In other words, we can generate an equation (whose terms are encoded in $\mathbf{a}$) that is linear in $\mathbf{x}$ that is satisfied if $\mathbf{x}$ is a good model. If we get N of these equations, we stack all of these together to make a matrix $\mathbf{A}$ with N rows. This gives a system of equations:

$$\mathbf{Ax} = \begin{bmatrix} - & \mathbf{a}_1 & - \\ & \vdots & \\ - & \mathbf{a}_N & - \end{bmatrix} \mathbf{x} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}. \tag{3}$$

Two things make this a little tricky. First, you can *always* trivially satisfy this by setting $\mathbf{x} = \mathbf{0}$. Second, if you have $p < N$, then things are overconstrained, so no non-zero vector will actually satisfy all of these equations. These difficulties are overcome by instead finding the unit vector $\mathbf{x} \in \mathbb{R}^p$ that minimizes $||\mathbf{Ax}||_2^2$. The unit vector $\mathbf{x}^*$ that minimizes $||\mathbf{Ax}||_2^2$ ends up being the eigenvector of $(\mathbf{A}^T\mathbf{A})$ with smallest eigenvalue.

Why Does This Work? If we expand out $||\mathbf{Ax}||_2^2$, we get $(\mathbf{Ax})^T(\mathbf{Ax})$ and then $\mathbf{x}^T(\mathbf{A}^T\mathbf{A})\mathbf{x}$. Our requirement that $||\mathbf{x}||_2 = 1$ just comes the fact that the value of $||\mathbf{Ax}||_2^2$ can be arbitrarily changed by scaling $\mathbf{x}$. So the thing we'd really like to optimize for accounts for this scale and is:

$$\frac{\mathbf{x}^T(\mathbf{A}^T\mathbf{A})\mathbf{x}}{\mathbf{x}^T\mathbf{x}} = \frac{\mathbf{x}^T(\mathbf{A}^T\mathbf{A})\mathbf{x}}{||\mathbf{x}||_2^2} = \frac{\mathbf{x}^T(\mathbf{A}^T\mathbf{A})\mathbf{x}}{||\mathbf{x}||_2 \cdot ||\mathbf{x}||_2} = \left(\frac{\mathbf{x}}{||\mathbf{x}||_2}\right)^T (\mathbf{A}^T\mathbf{A}) \left(\frac{\mathbf{x}}{||\mathbf{x}||_2}\right). \tag{4}$$

From here, check out the Rayleigh Quotient. This is a *useful* property of eigenvectors. If have a maximization problem, the vector that maximizes the property is – you guessed it – the eigenvector corresponding to the maximum eigenvalue.

Why Doesn't This work? While $\mathbf{x}$ really should satisfy $\mathbf{Ax} = \mathbf{0}$ if things are perfect, the value $||\mathbf{Ax}||_2^2 = \sum_{i=1}^N (\mathbf{a}_i^T\mathbf{x})^2$ often doesn't have any real meaning. So while the minimum of $||\mathbf{Ax}||_2^2$ is probably not a bad idea, it's not clear that making that value small corresponds to making the estimates of geometry better. So in practice, it's common to use homogeneous least-squares to get a good first guess that you polish off by using a non-linear optimizer on some other cost function (typically expressed as a distance between where points should be in 2D and where they are).

TL;DR: The too-long version is: we give you a design matrix $\mathbf{A}$ such that a model $\mathbf{x}$ that fits perfectly should satisfy $\mathbf{Ax} = \mathbf{0}$. You then compute the eigenvector corresponding to the smallest eigenvalue of $\mathbf{A}^T\mathbf{A}$. Alternatively, you can also get the last singular vector of the right-singular vectors of a SVD $\mathbf{U\Sigma V}^T = \mathbf{A}$ of $\mathbf{A}$ ($\mathbf{V}$ are eigenvectors of $\mathbf{A}^T\mathbf{A}$).

1.1.6 The Closest Rank-k Matrix

Given a matrix $\mathbf{M} \in \mathbb{R}^{n \times n}$, we often want to find some matrix $\hat{\mathbf{M}}_k$ that is as "close" to $\mathbf{M}$ as possible, but which has a lower rank (e.g., some k with $k < n$).

Rank Reminder: Recall that the rank of a matrix is the dimension of the space spanned by its vectors. Matrices of size $n \times n$ that are rank deficient do not span the full space of $\mathbb{R}^n$. Equivalently, they do not have an inverse and have a non-trivial nullspace.

We can solve for the closest lower rank matrix with an SVD. Suppose $\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T = \mathbf{M}$ is the singular value decomposition (SVD) of $\mathbf{M}$. The matrices $\mathbf{U}, \mathbf{V}^T$ are just rotation matrices, and $\mathbf{\Sigma}$ should be $\text{diag}([\sigma_1, \ldots, \sigma_n])$ with $\sigma_i \geq \sigma_{i+1}$. In other words, $\mathbf{\Sigma}$ is a diagonal matrix with the ascending singular values σ_i along the diagonal. Let's make $\mathbf{\Sigma}_k = \text{diag}([\sigma_1, \ldots, \sigma_k, 0, \ldots 0])$ by replacing σ_{k+1} and later singular values with zeros. Then, we can make a matrix $\hat{\mathbf{M}}_k = \mathbf{U}\mathbf{\Sigma}_k\mathbf{V}^T$, which consists of the original SVD of $\mathbf{M}$, but with all but the first k singular values set to 0.

One nice result is the Eckart–Young–Mirsky theorem, which tells us that this is not a ridiculous idea. Specifically, $\hat{\mathbf{M}}_k$ is actually the closest matrix of rank k to $\mathbf{M}$! In other words

$$\hat{\mathbf{M}}_k = \underset{\mathbf{M}' \in \mathbb{R}^{n \times n}, \text{rank}(\mathbf{M}')=k}{\arg\min} ||\mathbf{M}' - \mathbf{M}||. \tag{5}$$

I'm leaving the norm unspecified, because this holds for both (a) the Frobenius norm $|| \cdot ||_F$, which is a fancy name for taking the square root of the sum of the squares of all the entries; and (b) the spectral norm $|| \cdot ||_2$, which is the largest singular value (intuitively a measure of the maximum length a unit norm vector can have after being multiplied by the matrix).

1.1.7 Finding the Null Space

Given a $n \times n$ matrix $\mathbf{A}$ with reduced rank, you can find a basis for the nullspace by the SVD. In a particularly useful instance, suppose $\mathbf{A}$ has rank $n-1$, and if $\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$ is the singular value decomposition of $\mathbf{A}$. Then, the left nullspace of $\mathbf{A}$ is given by the last column of $\mathbf{U}$ and the right nullspace by the last column of $\mathbf{V}$.

You have to be careful with how linear algebra packages return results. Just as you have to be careful with whether the eigenvectors are rows or columns, you should be careful here too. If you think you have a solution $\mathbf{x}$ for the right nullspace, try computing $\mathbf{A}\mathbf{x}$; for the left nullspace, try computing $\mathbf{x}^T\mathbf{A}$. These should be close to the zero vector.

1.2 Linear Camera Calibration

If I have a set of N 2D points $\mathbf{p}_i \equiv [u_i, v_i, 1]$ and 3D points $\mathbf{X}_i$, I know that $\mathbf{p}_i \equiv \mathbf{M}\mathbf{X}_i$ should hold. This also means that $\mathbf{p}_i \times \mathbf{M}\mathbf{X}_i = 0$. You can use this to derive that:

$$\begin{bmatrix} \mathbf{0}^T & \mathbf{X}_i^T & -v_i\mathbf{X}_i^T \\ \mathbf{X}_i^T & \mathbf{0}^T & -u_i\mathbf{X}_i^T \\ -v_i\mathbf{X}_i^T & u_i\mathbf{X}_i^T & \mathbf{0}^T \end{bmatrix} \begin{bmatrix} \mathbf{M}_1^T \\ \mathbf{M}_2^T \\ \mathbf{M}_3^T \end{bmatrix} = \mathbf{0}, \tag{6}$$

where $\mathbf{M}_1$, $\mathbf{M}_2$, and $\mathbf{M}_3$ are the three rows of $\mathbf{M}$. If you re-derive this yourself, you may have to flip a sign. There are actually only two linearly independent equations in this matrix: you can see this by multiplying the top row by u_i and the middle row by $-v_i$ and then adding the two of them to the third row.

Given that you have a matrix of the form $\mathbf{A}\mathbf{x} = \mathbf{0}$, you're set to use homogeneous least-squares! It's probably best to just grab the first two rows for of the resulting matrix per 2D/3D correspondence.

1.3 The Essential Matrix

Suppose we have two views of a scene. If we have access to the intrinsics of our cameras, the Essential Matrix represents the relationship between the two views.

Setup: Suppose $\mathbf{K}$, $\mathbf{K}'$ are the intrinsics of images 1 and 2 and the mapping from camera 1's coordinate system to camera 2's is given by a rotation $\mathbf{R} \in \mathbb{R}^{3\times3}$ and translation $\mathbf{t} \in \mathbb{R}^3$. This means that the projection matrix for image 1 is $\mathbf{K}[\mathbf{I}, \mathbf{0}]$ and the projection matrix for image 2 is $\mathbf{K}'[\mathbf{R}, \mathbf{t}]$. We'll then assume we have a point in 3D $\mathbf{X}$ that projects to pixel location $\mathbf{p}$ in image 1 and pixel location $\mathbf{p}'$ in image 2, or $\mathbf{p} \equiv \mathbf{K}[\mathbf{I}, \mathbf{0}]\mathbf{X}$ and $\mathbf{p}' \equiv \mathbf{K}'[\mathbf{R}, \mathbf{t}]\mathbf{X}$.

Normalized Coordinates: Note that $\mathbf{p}, \mathbf{p}'$ are defined in image coordinates. Since they are 2D objects in the image plane, you cannot combine them with 3D objects in the world. However, if we undo the projection, or compute $\hat{\mathbf{p}} = \mathbf{K}^{-1}\mathbf{p}$ and $\hat{\mathbf{p}}' = \mathbf{K}'^{-1}\mathbf{p}'$, we now have the rays in 3D that go through the origins of cameras 1 and 2. Mechanically, you can now see that $\hat{\mathbf{p}} \equiv [\mathbf{I}, \mathbf{0}]\mathbf{X}$ and $\hat{\mathbf{p}}' \equiv [\mathbf{R}, \mathbf{t}]\mathbf{X}$. These geometrically meaningful representations of the coordinates are called *normalized coordinates*

One annoying detail is that $\hat{\mathbf{p}}$ is defined in camera 1's coordinate system and $\hat{\mathbf{p}}'$ is defined in camera 2's coordinate system. That said, we have the rotation and translation that transforms them. When working through this, be careful of what coordinate system each vector is in!

Deriving E: We'll start off with the rays $\hat{\mathbf{p}} = \mathbf{K}^{-1}\mathbf{p}$ and similarly for $\hat{\mathbf{p}}' = \mathbf{K}'^{-1}\mathbf{p}'$. We can convert the ray $\hat{\mathbf{p}}$ from camera 1 to camera 2's coordinate system by the matrix $\mathbf{R}$. Then, the following vectors must be all co-planar: $\mathbf{R}\hat{\mathbf{p}}$, $\hat{\mathbf{p}}'$ and $\mathbf{t}$. This is satisfied when their triple product $\hat{\mathbf{p}}'^T(\mathbf{t} \times \mathbf{R}\hat{\mathbf{p}}) = 0$. One can expand this out to form

$$\hat{\mathbf{p}}'^T(\mathbf{t} \times \mathbf{R}\hat{\mathbf{p}}) = 0 \quad \rightarrow \quad \hat{\mathbf{p}}'^T[\mathbf{t}]_\times\mathbf{R}\hat{\mathbf{p}} = 0. \tag{7}$$

The Essential Matrix $\mathbf{E}$ is $[\mathbf{t}]_\times\mathbf{R}$. Then, given any points $\hat{\mathbf{p}}, \hat{\mathbf{p}}'$ in *normalized coordinates*, they must always satisfy $\hat{\mathbf{p}}'^T\mathbf{E}\hat{\mathbf{p}} = 0$.

Writing E out explicitly: To point to how the Fundamental Matrix works, let's just write the full expression out from pixels on one side to pixels on the other. First, let's do it in a somewhat reasonably grouped setting:

$$(\mathbf{K}'^{-1}\mathbf{p}')^T\mathbf{E}(\mathbf{K}^{-1}\mathbf{p}) = 0, \tag{8}$$

which we can re-arrange a bit to make the nicely symmetric form

$$\mathbf{p}'^T\mathbf{K}'^{-T}\mathbf{E}\mathbf{K}^{-1}\mathbf{p} = 0. \tag{9}$$

Decomposing E: Given a $\mathbf{R}$, $\mathbf{t}$, there's only one $\mathbf{E}$ (up to a scale). On the other hand, given an $\mathbf{E}$ (for instance fit by correspondences), there are multiple $\mathbf{R}$ and $\mathbf{t}$ that could have produced it. However, since $\mathbf{E} = [\mathbf{t}]_\times\mathbf{R}$ with $\mathbf{R}$ a rotation matrix and $[\mathbf{t}]_\times$ being skew-symmetric, there are only a few options/well-behaved ambiguities. Specifically, one has to choose between one of two rotations and a sign for the translation. Finally, there's a fundamental scale ambiguity for $\mathbf{t}$.

1.4 The Fundamental Matrix

If we don't have information about the intrinsics of the camera, the Fundamental Matrix represents the relationship between the two cameras. What's exciting is that the Fundamental Matrix exists for any two images capturing the same scene and it constrains how the points could be related without ever reasoning about the 3D scene explicitly! If you can estimate the Fundamental Matrix from a few strong correspondences, then you can constrain all the other matches.

Specifically, suppose a point $\mathbf{p} \equiv [u, v, 1]^T$ in image 1 and a point $\mathbf{p}' \equiv [u', v', 1]^T$ in image 2 are both the result of projecting some 3D point $\mathbf{X}$ by projection matrices $\mathbf{M}_1$ and $\mathbf{M}_2$. In other words $\mathbf{p} \equiv \mathbf{M}_1\mathbf{X}$ and $\mathbf{p}' \equiv \mathbf{M}_2\mathbf{X}$. Then the Fundamental Matrix $\mathbf{F}$ between the two cameras satisfies the relationship $\mathbf{p}'^T\mathbf{F}\mathbf{p} = 0$,

or expanding out a few times

$$\mathbf{p}'^T\mathbf{F}\mathbf{p} = 0 \rightarrow \begin{bmatrix} u' & v' & 1 \end{bmatrix} \mathbf{F} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = 0 \rightarrow \begin{bmatrix} u' & v' & 1 \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = 0. \quad (10)$$

You can multiply $\mathbf{Fp}$ explicitly to get find an expression as the dot product between two vectors, or

$$[u', v', 1] \begin{bmatrix} uf_{11} + vf_{12} + f_{13} \\ uf_{21} + vf_{22} + f_{23} \\ uf_{31} + vf_{32} + f_{33} \end{bmatrix} = 0, \quad (11)$$

and then multiplying further you get

$$u'uf_{11} + u'vf_{12} + u'f_{13} + v'uf_{21} + v'vf_{22} + v'f_{23} + uf_{31} + vf_{32} + f_{33} = 0. \quad (12)$$

You can pull out all of the entries in $\mathbf{F}$ to yield

$$[u'u, u'v, u', v'u, v'v, v', u, v, 1][f_{11}, f_{12}, f_{13}, f_{21}, f_{22}, f_{23}, f_{31}, f_{32}, f_{33}]^T = 0. \quad (13)$$

This expression is not particularly enlightening, but is useful for fitting the matrix.

Fundamental Matrix Facts: $\mathbf{F}$ has a number of useful properties. Again, suppose a point $\mathbf{p} \equiv [u, v, 1]^T$ in image 1 and a point $\mathbf{p}' \equiv [u', v', 1]^T$ in image 2 are both the result of projecting some 3D point $\mathbf{X}$ by projection matrices $\mathbf{M}_1$ and $\mathbf{M}_2$. Note, however, we do not *need* $\mathbf{M}_1$ and $\mathbf{M}_2$ to reason via $\mathbf{F}$.

1. **Relationship to Essential Matrix**: Given an Essential Matrix $\mathbf{E}$ and camera intrinsics $\mathbf{K}$ and $\mathbf{K}'$, then the following relationship holds

   $$\mathbf{F} = \mathbf{K}'^{-T}\mathbf{E}\mathbf{K}^{-1}. \quad (14)$$

   One can think of this equation as operationally describing how the Fundamental Matrix works: it is the Essential Matrix wrapped in intrinsic matrices that convert the pixel locations (i.e., in grid coordinates) into honest to goodness 3D rays (i.e., x/y/z in the real world).

   If we break down everything, we get the involved expression explaining how pixel locations $\mathbf{p}$ and $\mathbf{p}'$ must be related:

   $$\mathbf{p}'\mathbf{K}'^{-T}[\mathbf{t}]_x\mathbf{R}\mathbf{K}^{-1}\mathbf{p} = 0. \quad (15)$$

   This expression suggests that you won't be able to uniquely decompose $\mathbf{F}$ into something clean: there are a lot of matrices in this expression so changes in one can compensate for changes in another.

2. **Epipolar Lines**: If we have a point $\mathbf{p}$ in image 1, we don't actually know in 3D that point came from. We just know that it corresponds to a ray going out into the world. This ray, when projected back to image 2 results in a *line* – recall that projection preserves lines – called an epipolar line. Note that without the intrinsics, we do not know where this ray is; we only know that it exists.

   Now, let's look at generating that epipolar line from the Fundamental Matrix $\mathbf{F}$. You can break up $\mathbf{p}'^T\mathbf{F}\mathbf{p} = 0$ in two ways: $\mathbf{p}'^T(\mathbf{F}\mathbf{p}) = 0$ and $(\mathbf{p}'^T\mathbf{F})\mathbf{p} = 0$.

   Let's look at the first one. Given that $\mathbf{p}'$ is a point, $(\mathbf{F}\mathbf{p})$ can be thought of as the coefficients of an equation of a line: $\mathbf{p}'^T(\mathbf{F}\mathbf{p}) = 0$ is a way of expressing that that the point $\mathbf{p}'$ is on the line $\mathbf{F}\mathbf{p}$. Thus, the epipolar line for $\mathbf{p}$ (i.e., the line that $\mathbf{p}'$ must lie on) is given by $\mathbf{F}\mathbf{p}$.

   The second way of writing things, $(\mathbf{p}'^T\mathbf{F})\mathbf{p} = 0$ can be cleaned up a bit by transposing, yielding $\mathbf{p}^T(\mathbf{F}^T\mathbf{p}') = 0$. This can then be seen as taking the point $\mathbf{p}$ and checking if it lies on the line $(\mathbf{F}^T\mathbf{p}')$.

3. **Epipoles:** The epipoles are the projection of baseline between the two cameras into the images and accordingly the projection of the each camera's origin into the other's. Suppose we know that we have a point $\mathbf{p}$ in image 1. Geometrically speaking, this point $\mathbf{p}$ could result from $\mathbf{X}$ being at the origin of camera 1. Thus, the epipolar line in image 2 must always contain the epipole.

   Thus, the epipoles are also where all the epipolar lines intersect. This somewhat innocuous statement implies something quite interesting about the nullspaces of $\mathbf{F}$. Specifically, suppose $\mathbf{e}$ is the epipole in image 1. Then, for *every* single point $\mathbf{p}'$, if I construct its epipolar line $\mathbf{p}'^T\mathbf{F}$ (i.e., the equation of the line), $\mathbf{e}$ lies on this line *every* single time. This means that $\mathbf{p}'^T\mathbf{F}\mathbf{e} = 0$ irrespective of how I pick $\mathbf{p}'$. The only way this is true is if $\mathbf{F}\mathbf{e} = \mathbf{0}$.

   Thus, the epipole in image 1 $\mathbf{e}$ is the right-nullspace of $\mathbf{F}$ and the epipole in image 2 (by similar logic) is the left-nullspace of $\mathbf{F}$. Note that if you compute the nullspace, you will usually get a unit vector from the software you're using. To get the real coordinates of the epipole, you'll have to scale things so the last entry is one.

4. **Rank of F:** Given that $\mathbf{F}$ has non-trivial nullspaces (namely the epipoles), $\mathbf{F}$ is rank deficient. The deduction that $\mathbf{F}$ is rank deficient comes from those usual long lists about equivalent statements in linear algebra about matrices: full rank matrices are invertible, only have trivial nullspaces (i.e., $\mathbf{0}$), etc. etc. In particular, $\mathbf{F}$ has rank 2 since the nullspace is 1D (all the scaled copies of $\mathbf{e}$ for right and $\mathbf{e}'$ for left).

1.5 Solving for F

If we have pairs of points in correspondence $[u_i, v_i] \leftrightarrow [u'_i, v'_i]$, then we can make one row of a matrix per correspondence, taking the form

$$\mathbf{U} = \begin{bmatrix} & & & & \vdots & & & & \\ u_i u'_i & u_i v'_i & u_i & v_i u'_i & v_i v'_i & v_i & u'_i & v'_i & 1 \\ & & & & \vdots & & & & \end{bmatrix}. \tag{16}$$

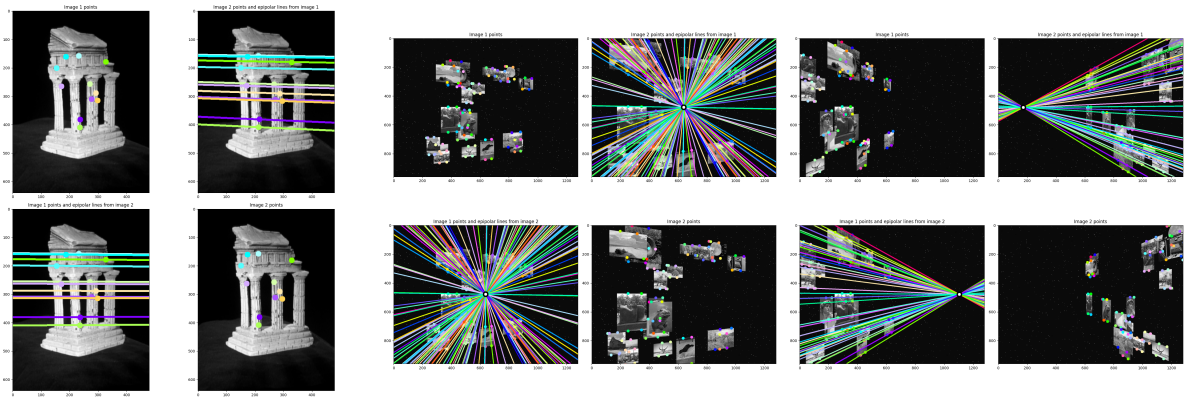
This can be solved via standard homogeneous least-squares: if $\mathbf{f}$ contains all the parameters of $\mathbf{F}$, then $\mathbf{U}\mathbf{f}$ should be $\mathbf{0}$. We thus solve for $\mathbf{f}$ by finding the eigenvector with smallest eigenvalue of $\mathbf{U}^T\mathbf{U}$.

Complication 1 – Rank Deficiency: The Fundamental Matrix is not full rank and the optimization will possibly give us a full rank matrix $\mathbf{F}$. So you can compute the closest low-rank matrix using SVD. The thing that goes wrong with a full rank "Fundamentalish matrix" $\mathbf{F}'$ is that because it is full rank, it has no non-trivial nullspaces. Therefore, the epipoles don't exist and so the epipolar lines won't converge!

Complication 2 – Numerical Stability: This algorithm is a terrible idea numerically on raw pixel locations. So in practice, you need to rescale the data. The "In defense of the eight-point algorithm" paper has a somewhat involved solution. You can also just make a homography $\mathbf{T}$ that scales things so that the center is zero and the image is one unit wide. It's a good idea to: (1) pre-compute the transformations as a matrix $\mathbf{T}$, (2) apply it to the data, (3) fit the fundamental matrix $\mathbf{F}$ on the transformed data, (4) finally return $\mathbf{T}^T\mathbf{F}\mathbf{T}$. One can think of this form as just applying the transformation to both input points.

The numerical instability comes from the fact that the the matrix $\mathbf{U}$ contains entries like $u'u$, u and 1. If the image is of size $1,000 \times 1,000$, then the pixel coordinates u and u' are of size $\sim 10^3$. Then $u'u \sim 10^6$, $u \sim 10^3$ and $1 \sim 10^0$. Things are made worse if one tries to compute $\mathbf{U}^T\mathbf{U}$: the range is then $10^{12}, \ldots, 10^0$. By rescaling things, the relative gap is much smaller.

2 Camera Calibration



Temple zrtrans reallyInwards

Figure 1: Epipolar lines for some of the datasets. The Epipoles are marked in little white circles. Best seen by zooming in.

Task 1: Estimating M [20 points]

We will give you a set of 3D points $\{\mathbf{X}_i\}_i$ and corresponding 2D points $\{\mathbf{p}_i\}_i$. The goal is to compute the projection matrix $\mathbf{M}$ that maps from world 3D coordinates to 2D image coordinates. Recall that

$$\mathbf{p} \equiv \mathbf{M}\mathbf{X}, \tag{17}$$

and (see foreword) by deriving an optimization problem, you can solve for $\mathbf{M}$. The script `task1.py` shows you how to load the data. The data we want you to use is in `task1/`, but we show you how to use data from Task 2 and 3 as well. **Credit:** The data from task 1 and an early version of the problem comes from James Hays's Georgia Tech CS 6476.

(a) (5 points) **Fill in `find_projection` in `task1.py`.**

(b) (5 points) **Report $\mathbf{M}$** for the data in `task1/`.

(c) (5 points) **Describe how well the projection maps the points $\mathbf{X}_i$ to $\mathbf{p}_i$.** Specifically, compute the average distance in the image plane (i.e., pixel locations) between the homogeneous points $\mathbf{M}\mathbf{X}_i$ and 2D image coordinates $\mathbf{p}_i$, or

$$\frac{1}{N}\sum_{i}^{N} ||\text{proj}(\mathbf{M}, \mathbf{X}_i) - \mathbf{p}_i||_2. \tag{18}$$

(d) (5 points) **Describe what relationship, if any, there is between Equations 18 and 6.** Note that the points we've given you are well-described by a linear projection – there's no noise in the measurements – but in practice, there will be an error that has to minimize. Both equations represent objectives that could be used. If they are the same, show it; if they are not the same, report which one makes more sense to minimize. Things to consider include whether the equations directly represent anything meaningful.

3 Estimation of the Fundamental Matrix and Reconstruction

Data: we give you a series of datasets that are nicely bundled in the folder `task23/`. Each dataset contains two images `img1.png` and `img2.png` and a numpy file `data.npz` containing a whole bunch of variables. The script `task23.py` shows how to load the data.

Credit: `temple` comes from Middlebury's Multiview Stereo dataset. The images shown in the synthetic images are described in HW0's credits.

Task 2: Estimating F [50 points]

(a) (20 points) **Fill in** `find_fundamental_matrix` in `task23.py`. You should implement the eight-point algorithm. Remember to normalize the data (either the fancier approach works) or scaling by the image size and centering at 0 is fine for our purposes, and to reduce the rank of $\mathbf{F}$. You can compare with `cv2.findFundamentalMat(pts1,pts2,cv2.FM_8POINT)`, which should roughly match.

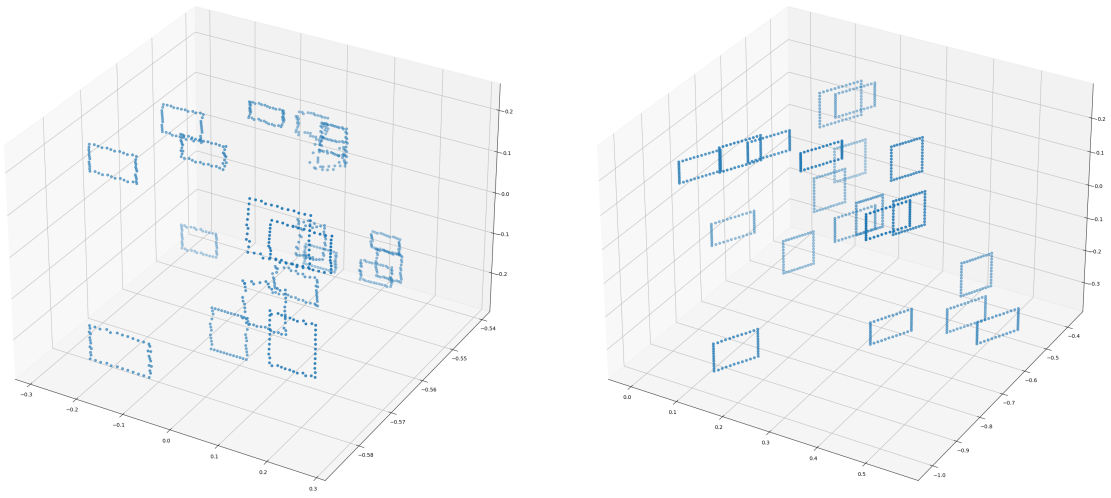
(b) (5 points) **Report F (normalized so $F_{2,2} = 1$) for** `temple`**,** `ztrans`**, and** `xtrans`**.**

(c) (10 points) **Fill in** `compute_epipoles`**.** This should return the homogeneous coordinates of the epipoles – remember they can be infinitely far away!

(d) (5 points) **Show epipolar lines for** `temple`**,** `reallyInwards`**, and another dataset of your choice.**

(e) (5 points) **Report the epipoles for** `reallyInwards` **and** `xtrans`.

Task 3: Triangulating X [30 points]



ztrans reallyInwards

Figure 2: Visualizations of reconstructions

The next step is extracting 3D points from 2D points and camera matrices, which is called triangulation. Let $\mathbf{X}$ be a point in 3D.

$$\mathbf{p} = \mathbf{M}_1\mathbf{X} \quad \mathbf{p}' = \mathbf{M}_2\mathbf{X} \tag{19}$$

Triangulation solves for $\mathbf{X}$ given $\mathbf{p}, \mathbf{p}', \mathbf{M}_1, \mathbf{M}_2$. We'll use OpenCV's algorithms to do this. You can feel free to use OpenCV's Fundamental Matrix code if you'd like for this part.

1. (5 points) **Compute and report the Essential Matrix E given the Fundamental Matrix F.** You should do this for the dataset `reallyInwards`. Recall that

$$\mathbf{F} = \mathbf{K}'^{-T}\mathbf{E}\mathbf{K}^{-1} \tag{20}$$

   and that $\mathbf{K}, \mathbf{K}'$ are always invertible (for reasonable cameras), so you can compute $\mathbf{E}$ straightforwardly.

2. (15 points) **Fill in `find_triangulation` in `task23.py`.**

   The first camera's projection matrix is $\mathbf{K}[\mathbf{I}, \mathbf{0}]$. The second camera's projection matrix can be obtained by decomposing $\mathbf{E}$ into a rotation and translation via `cv2.decomposeEssentialMat`. This function returns two matrices $\mathbf{R}_1$ and $\mathbf{R}_2$ and a translation $\mathbf{t}$. The four possible camera matrices for $\mathbf{M}_2$ are:

$$\mathbf{M}_2^1 = \mathbf{K}'[\mathbf{R}_1, \mathbf{t}], \quad \mathbf{M}_2^2 = \mathbf{K}'[\mathbf{R}_1, -\mathbf{t}], \quad \mathbf{M}_2^3 = \mathbf{K}'[\mathbf{R}_2, \mathbf{t}], \quad \mathbf{M}_2^4 = \mathbf{K}'[\mathbf{R}_2, -\mathbf{t}] \tag{21}$$

   You can identify which projection is correct by picking the one for which the most 3D points are in front of both cameras. This can be done by checking for the positive depth, which can be done by looking at the last entry of the homogeneous coordinate: the extrinsics put the 3D point in the camera's frame, where $z < 0$ is behind the camera, and the last row of $\mathbf{K}$ is $[0, 0, 1]$ so this does not change things.

   Finally, triangulate the 2D points using `cv2.triangulatePoints`.

3. (10 points) **Put a visualization of the point cloud for `reallyInwards` in your report.** You can use `visualize_pcd` in `utils.py` or implement your own.

References and Credits

- Temple dataset used in Tasks 2 and 3: http://vision.middlebury.edu/mview/data/.
- Part of the homework are taken from Georgia Tech CS 6476 by James Hays and CMU 16-385. Please feel free to similarly re-use our problems while similarly crediting us.