

ECEN 4532 - Lab 3: Perspective Transformations and Motion Tracking

Andrew Teta

February 25, 2019



Contents

Introduction	3
1 Introduction	3
1.1 Background	3
2 Perspective Distortion Correction	3
2.1 Linear Regression	3
2.2 Perspective Correction	5

1 Introduction

In this lab, we explore the low-level implementation of perspective distortion correction, and a simplified model of video motion tracking between frames in Python. We will mostly use `numpy` to manipulate N-D arrays of pixel information. Additionally, I used a free photo editing software called GIMP to select pixel indexes.

1.1 Background

Our algorithm for perspective distortion correction relies on the math theory of linear algebra, using a linear transformation matrix to map pixels from one image to another. The process of motion tracking between frames will be based on a series of image pyramids, using block-based motion estimation to "search" for a matching region in the next frame.

2 Perspective Distortion Correction

2.1 Linear Regression

We begin by defining linear regression as one approach to modeling the relationship between a set of dependent and independent variables. This method is commonly used to fit a model to an **over-determined** set of data points. By over-determined, we mean there are more equations defining the data set than there are variables, often occurring when many measurements are performed to estimate a small number of parameters.

Mathematically, we have a relationship of the form, $Ax = c$, where A is over-determined. This only has a solution if c lies in the column space of A , however there will not be an exact solution if A is over-determined and we instead seek an x that minimizes the mean-squared-error (MSE), defined as, $E = \|Ax - c\|^2$. Ultimately, we want to find a "pseudo-inverse", A^+ , such that $x^{\$} = A^+c$.

For a given point, c , we want to find the closest point in the column space of A . This turns out to be the projection, p , of c onto the column space. Thus, the error is the distance vector, $c - p$, orthogonal to the column space. To be orthogonal, the error vector must be orthogonal to every column of A .

$$\begin{aligned} (c - p)^T A &= 0 \\ p^T A &= c^T A \\ A^T p &= A^T c \end{aligned}$$

We are in search of $x^{\$}$ such that $Ax^{\$} = p$. Assuming the columns of A are linearly independent and substituting, we obtain

$$\begin{aligned} A^T Ax^{\$} &= A^T c \\ x^{\$} &= (A^T A)^{-1} A^T c \end{aligned}$$

Therefore,

$$A^+ = (A^T A)^{-1} A^T \tag{1}$$

To demonstrate linear regression, we consider a hypothetical experiment trying to correlate electrode sensor data to a subjective hunger measurement. The data is as follows.

Electrode 1 (mV)	Electrode 2 (mV)	Electrode 3 (mV)		Subjective Hunger
1	8	3		65.66
-46	-98	108		-1763.1
5	12	-9		195.2
63	345	-27		3625
23	78	45		716.9
-12	56	-8		339
1	34	78		-25.5
56	123	-5		1677.1

Figure 2: Experimental data

In Python, $A_p = A^+$ can be written as `A_p = np.linalg.inv(A.T.dot(A)).dot(A.T)` and calculating \mathbf{x}^s , `x = A_p.dot(c)`, the prediction coefficients, we have

$$x^s = \begin{bmatrix} x1 \\ x2 \\ x3 \end{bmatrix} = \begin{bmatrix} 11.93 \\ 8.02 \\ -3.98 \end{bmatrix} \quad (2)$$

Calculating the MSE for this prediction vector is simple, and results in

$$MSE = \begin{bmatrix} 2.36 \\ 2.31 \\ 12.22 \\ 0.05 \\ 13.41 \\ 1.73 \\ 0.36 \\ 8.56 \end{bmatrix} \quad (3)$$

2.2 Perspective Correction

We will use the concept of linear regression to correct perspective distortion in an image. Consider the following diagrams representing a simple model for imaging and projection.

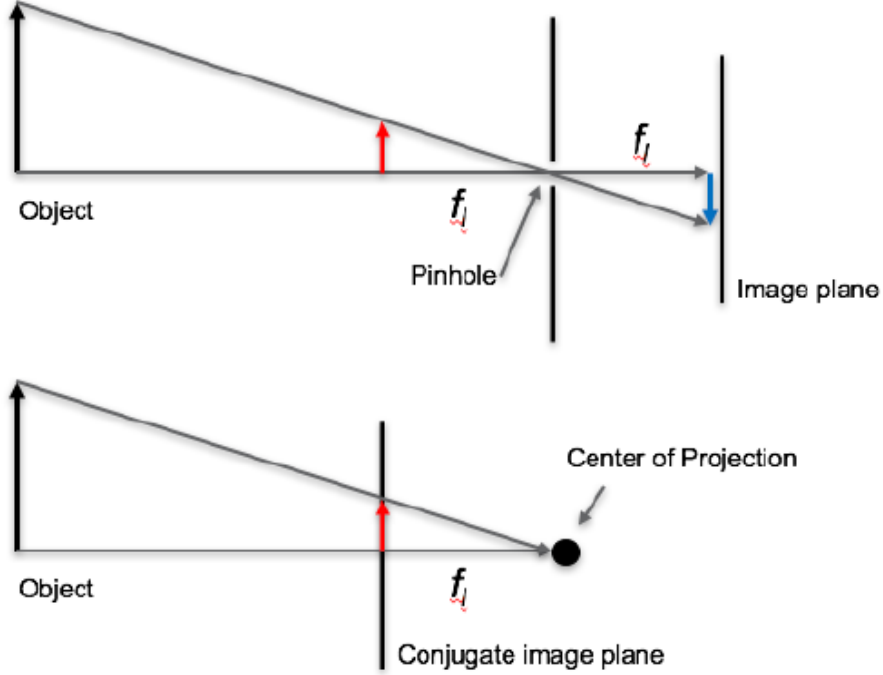


Figure 3: Simple projection imaging ray diagram.

Analyzing the relationships between points in the ray diagram of fig. 3, we see in fig. 4 there is a simple relationship for points in a projection.

Geometrically, we have

$$\frac{f}{X_3} = \frac{x_1}{X_1}$$

$$x_1 = f \frac{X_1}{X_3}$$

and by a similar argument,

$$x_2 = f \frac{X_2}{X_3}$$

Therefore, the camera coordinates, (X_1, X_2, X_3) are related to the canonical image coordinates, (x_1, x_2) as

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} f & 0 \\ 0 & f \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} \frac{1}{X_3} \quad (4)$$

However, this relationship is non-linear and we can do this another way. By finding a relationship in *homogeneous* coordinates, we arrive at a linear relationship

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ 1 \end{bmatrix} \quad (5)$$

where the canonical coordinates of the projection point, \vec{x} are found as x_1/x_3 and x_2/x_3 .

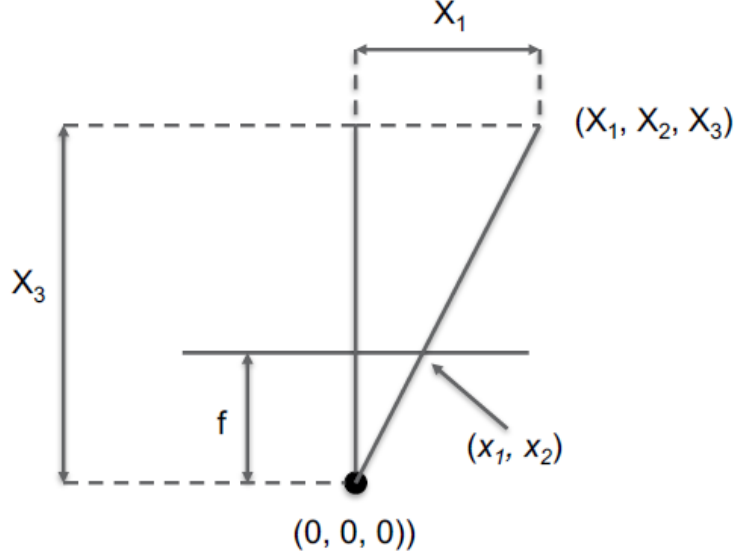


Figure 4: A projection image and arbitrary object plane of focal distance, f .

Thus, while in Euclidean geometry, a point in N -D space is described by an N -D vector, in projective geometry, a point in N -D space is described by an $(N+1)$ -D vector. The last coordinate, k is a multiplier of the first N coordinates. Then, a point can be scaled by k as $(k * rows, k * cols, k)$, a homogeneous vector.

Fact: Given a set of world points known to lie on a plane, there exists a matrix, H , that maps the (row, col) image points in one camera to the (row, col) image points in another camera. Mathematically,

$$\vec{v} = H\vec{x}. \quad (6)$$

For our purpose of correcting perspective distortion, we take \vec{x} to be a homogeneous vector of a point in the distorted image, \vec{v} to be a homogeneous vector of the same point in the corrected image, and seek a matrix H that maps lines converging at infinity to parallel lines in the Euclidean plane.

The process of perspective distortion correction will consist of choosing a set of points in a distorted image that lie on a plane, although not all on the same line. We will choose 20 points and two lines (10 points per line). Then, we will define 20 points, again in two lines, of which to map each pixel index.

Procedure Using GIMP and the mouse cursor, I selected 10 points on successive corners of the upper windows in the image shown in fig. 5 and 10 more points on the lower windows. Then choosing a relatively arbitrary range of values along the *col* dimension, and two constant values of *row*, I generated a set of 20 points in the horizontal direction, of which to map the original points into a corrected image.



Figure 5: Distorted image to be corrected.

Thus, the mapping for the first line is

$$\begin{array}{cc}
 \begin{bmatrix} 750 & 215 \\ 675 & 235 \\ 625 & 250 \\ 575 & 265 \\ 535 & 275 \\ 500 & 285 \\ 475 & 295 \\ 445 & 300 \\ 425 & 305 \\ 405 & 315 \end{bmatrix} & \longrightarrow & \begin{bmatrix} 750 & 305 \\ 711.67 & 305 \\ 673.33 & 305 \\ 635 & 305 \\ 596.67 & 305 \\ 558.33 & 305 \\ 520 & 305 \\ 481.67 & 305 \\ 443.33 & 305 \\ 405 & 305 \end{bmatrix}
 \end{array} \tag{7}$$

and for the second line,

$$\begin{bmatrix} 745 & 575 \\ 670 & 555 \\ 615 & 540 \\ 565 & 525 \\ 530 & 515 \\ 495 & 505 \\ 468 & 500 \\ 440 & 495 \\ 420 & 488 \\ 400 & 483 \end{bmatrix} \longrightarrow \begin{bmatrix} 750 & 475 \\ 711.67 & 475 \\ 673.33 & 475 \\ 635 & 475 \\ 596.67 & 475 \\ 558.33 & 475 \\ 520 & 475 \\ 481.67 & 475 \\ 443.33 & 475 \\ 405 & 475 \end{bmatrix} \quad (8)$$

Considering a point in the distorted image to be (c, d) and a point in the corrected image, (a, b) , the above relation can be expressed

$$\begin{aligned}
(c_1, d_1) &= (a_1, b_1) \\
(c_2, d_2) &= (a_2, b_2) \\
&\dots \\
(c_{20}, d_{20}) &= (a_{20}, b_{20})
\end{aligned}$$

Now, the relationship we need to find is

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix} \begin{bmatrix} c \\ d \\ 1 \end{bmatrix} \text{ and } \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} v_1/v_3 \\ v_2/v_3 \end{bmatrix} \quad (9)$$

Solving for v_1/v_3 and v_2/v_3 in 9, we find

$$\begin{aligned}
a &= \frac{h_{11}c + h_{12}d + h_{13}}{h_{31}c + h_{32}d + 1} \\
b &= \frac{h_{21}c + h_{22}d + h_{23}}{h_{31}c + h_{32}d + 1}
\end{aligned}$$

which can also be written in matrix form as

$$\begin{bmatrix} c & d & 1 & 0 & 0 & 0 & -ac & -ad \\ 0 & 0 & 0 & c & d & 1 & -bc & -bd \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix} \quad (10)$$

The Python procedure for this process goes as follows. First, we define a list of pixel coordinate points in the distorted image, `inputPoints` and the points to map into, `outputPoints`. Then, we build the left-hand side of (10), constructing a $[40, 8]$ matrix, `mapping` (two rows for each point). Using the same method as we did in the linear regression problem, we find the pseudo inverse, `mapping_inv = np.linalg.inv(mapping.T.dot(mapping)).dot(mapping.T)`. Reshaping the set of output points into a $[40, 1]$ vector allows us to calculate $H = mapping_inv \cdot outputPoints$ ($[9, 1]$). We then append 1 to the end, and reshape into a $[3, 3]$ matrix.

At this point, we are ready to remap every pixel in the distorted image to generate a corrected image. We initialize a new `numpy` array to be the same dimensions as the distorted image and begin stepping over each pixel. The method, now, is to compute $\vec{v} = H^{-1} \cdot \vec{p}$, where \vec{p} is the pixel index of the corrected image, and $[a, b] = [v_1/v_3, v_2/v_3]$

We then check to verify that $[a, b]$ is within the bounds of the distorted image dimensions and perform a bilinear interpolation to find the RGB intensity value for the current pixel in the corrected image.



(a) Original "distorted" image.



(b) Corrected image.

Figure 6: Perspective distortion correction.

Notice how the image gets increasingly blurry toward the left-hand side. This is due to the interpolation in this region. The mapping we attempted, tried to squeeze the right-hand side down, while stretching the left-hand side up. The mapping function also sets pixels to black if they are out of range of the original image, so there are large black triangles, where the perspective correction had no data to interpolate from for those regions. It may look better if the image was cropped, or a different set of output points was chosen. Unfortunately, this method also does not accommodate points that do not lie in the same plane as those being remapped very well. Notice the vines at the bottom left of the original image and how they were stretched in processing. This makes sense if we imagine the vines do live in the same plane as the side of the building, like graffiti painted on. Then, the vines appear to spread over a large horizontal length of the wall and when mapped, appear stretched.

Choosing a different set of points in the output image, the picture in fig. 7 was generated. Notice how it is possible to effectively zoom in using this remapping method.



Figure 7: A different mapping.