

ECE661: Computer Vision

Homework 2

Bruce Coburn
coburn6@purdue.edu

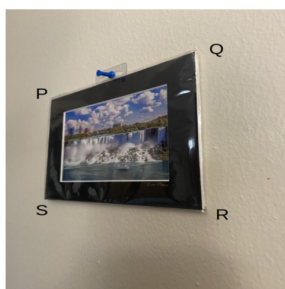
Task-1

We are given four images: three which consist of different views of a wall-mounted photo-frame (Figures 1(a-c)) while one is of Alex Honnold (Figure 1(d)) climbing El Capitan without a rope or safety equipment (becoming the first person to “free solo” the route). Our subtasks are to complete the following:

1. Using just four points, pick a region of interest (ROI) from the image of Alex Honnold and project that region onto the PQRS regions in Figures 1(a-c).
2. Find homographies between Figures 1(a-b) and Figures 1(b-c), then multiply the resultant homography to Figure 1(a). The resulting image should look similar to the image shown in Figure 1(c).
3. Using only affine homographies, map Alex’s image to the PQRS regions in all photo-frame images. Report the results. If a result is better than the others, explain why.



(a) View-1



(b) View-2



(c) View-3



(d) Image to be mapped:
Alex Honnold

Figure 1: Images for Task-1. Note that both outer or inner corner points of the given photo-frame images are acceptable solutions.

Task-1.1: Logic

We are interested in finding the homography which will transform points in our *domain* to a particular *range*. To do so, we manually annotated pixel coordinates that served as correspondence points between our domain and range images. For reference, our domain image points will have the following homogeneous representation:

$$\mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad (1)$$

Whereas our range image points (or transformed points) will have the following homogeneous representation:

$$\mathbf{X}' = \begin{bmatrix} x'_1 \\ x'_2 \\ x'_3 \end{bmatrix} \quad (2)$$

We do know that the relationship between our domain and image points is:

$$\mathbf{X}' = \mathbf{H}\mathbf{X} \quad (3)$$

Where $\mathbf{H}$ is a 3x3 non-singular matrix that can be represented as:

$$\mathbf{H} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \quad (4)$$

Noting all of the above equations together and substituting into Equation 3, we arrive at the following expression:

$$\begin{bmatrix} x'_1 \\ x'_2 \\ x'_3 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad (5)$$

We can then derive individual expressions for each of our range image points ($\mathbf{X}'$):

$$x'_1 = h_{11}x_1 + h_{12}x_2 + h_{13}x_3$$

$$x'_2 = h_{21}x_1 + h_{22}x_2 + h_{23}x_3$$

$$x'_3 = h_{31}x_1 + h_{32}x_2 + h_{33}x_3$$

Ultimately, we are interested in the physical point representation of our transformed points; meaning we need to convert our range image points to $\mathbb{R}^2$ representation. To do so, we divide our x'_1 and x'_2 by x'_3 to get our range physical points, (x', y') :

$$x' = \frac{x'_1}{x'_3} = \frac{h_{11}x_1 + h_{12}x_2 + h_{13}x_3}{h_{31}x_1 + h_{32}x_2 + h_{33}x_3}$$

$$y' = \frac{x'_2}{x'_3} = \frac{h_{21}x_1 + h_{22}x_2 + h_{23}x_3}{h_{31}x_1 + h_{32}x_2 + h_{33}x_3}$$

Because we are working homogeneous structures, we are only concerned about the ratios. This simplifies our range physical point expressions because we are free to arbitrarily set $h_{33} = 1$ and divide by x_3 . Finally, we get our final range physical point expressions:

$$x' = \frac{h_{11}x_1 + h_{12}x_2 + h_{13}}{h_{31}x_1 + h_{32}x_2 + 1}$$

$$y' = \frac{h_{21}x_1 + h_{22}x_2 + h_{23}}{h_{31}x_1 + h_{32}x_2 + h_{33}}$$

If we rearrange our expressions, we arrive at the following:

$$xh_{11} + yh_{12} + h_{13} - xx'h_{31} - yx'h_{32} - x' = 0 \quad (6)$$

$$xh_{21} + yh_{22} + h_{23} - xy'h_{31} - yy'h_{32} - y' = 0 \quad (7)$$

In Equations 6 and 7, we can observe that we have 8 unknowns (h_{ij}) amongst 2 equations. However, our 8 unknowns can be solved with 4 pairs of correspondence points. One pair of correspondence points gives one domain point ($[x_i, y_i]$) and one range point ($[x'_i, y'_i]$), meaning that each pair of correspondence points gives us 2 equations and we only need 4 correspondence points to alleviate our 8 unknowns. The matrix representation for our system of equations (using 4 correspondence points) is the following:

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x'_1 & -y_1x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y'_1 & -y_1y'_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2x'_2 & -y_2x'_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2y'_2 & -y_2y'_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3x'_3 & -y_3x'_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3y'_3 & -y_3y'_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4x'_4 & -y_4x'_4 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -x_4y'_4 & -y_4y'_4 \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} = \begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \\ x'_4 \\ y'_4 \end{bmatrix} \quad (8)$$

To make this representation more readable, we will simplify Equation 8 to be:

$$\mathbf{A}\mathbf{h} = \mathbf{X}'_{\text{corr}} \quad (9)$$

Since we are interested in the parameters, $\mathbf{h}$, of our general homography matrix, $\mathbf{H}$, we can see that:

$$\mathbf{h} = \mathbf{A}^{-1}\mathbf{X}'_{\text{corr}} \quad (10)$$

Where we can then reshape $\mathbf{h}$ into our 3x3 non-singular homography matrix, $\mathbf{H}$; matching its form in Equation 5. Now, to transform our domain points to our range points, we simply follow the operation in Equation 3.

Task-1.1: Implementation

The python code is meant to be (relatively) modular and efficient since this assignment involves a lot of the same operations over a handful of different images. First, we need to address how pixel coordinates (for corresponding PQRS regions) were extracted.

The file, `retrieve_pixel_coordinates.py`, allows us to interactively select points on images, annotate corresponding P, Q, R, S points and store both the pixel coordinate information and newly annotated images to a local directory. It should be noted that pixel coordinate information is stored in `img_data.json`. Heavy inspiration came from an online post [2], which served as the GUI foundation for the final implementation.

Once we have written pixel coordinate information to `img_data.json`, we can execute `toplevel_hw2.py` which handles the code implementation of the logic presented in the prior section. The code has been commented and laid out in, hopefully, a straightforward manner but there are a handful of considerations that were taken.

To solve for our general homography, $\mathbf{H}$, the functions `solve_and_perform_homography` and `solve_homography` were created. The numpy function, `numpy.linalg.solve`, extracts the solutions for our general homography matrix, $\mathbf{H}$, when we input a corresponding unknowns matrix, $\mathbf{A}$, and the rasterized vector of our correspondence points, $\mathbf{X}'_{\text{corr}}$. According to the Numpy documentation for `numpy.linalg.solve` [5], we are able to use this method because our linear matrix expression (Equation 9) is well determined and full rank.

Now that we have our general homography matrix, $\mathbf{H}$, we can implement Equation 5 once we reshape $\mathbf{h}$. To do so, and favoring efficiency, we utilized NumPy's ability to *vectorize* operations. In particular, vectorized operations are the process of applying an operation to an entire array, rather than iterating over individual elements using loops. In our case, we would have to implement nested `for` loops which would iterate over every pixel element in an image array. Nested `for` loops were taking a significantly long time so we decided to implement vectorized operations instead. This can be observed in the `perform_homography` function.

It should also be noted that instead of directly transforming points from the domain image to the range image, an inverse mapping method was adopted. Initially, we were having issues assigning valid values to our transformed image when taking a forward mapping approach. This inverse mapping method involves computing the transformed coordinates in the domain image that correspond to each point in the range image. This helps to ensure that all locations in the range image are assigned valid values.

As explained by Gonzalez and Woods in their book *Digital Image Processing* [3], inverse mapping is more efficient and widely used in practice because it avoids gaps or unassigned pixels in the output image. The book discusses that by scanning the output pixel locations and computing corresponding locations in the input image using the inverse transformation matrix, all output pixels are properly assigned, which aligns with our approach. The book also mentions interpolation, where we simply rounded computed values to the closest integer using `numpy rint()` [6]; effectively nearest-neighbor interpolation. We then normalize the third coordinate back to 1 to align with HC representation and we filter out points that fall outside of the image bounds.

Task-1.1: Results

Below are the annotated pixel coordinates for our range images:



Figure 2: Annotated Coordinates for img1



Figure 3: Annotated Coordinates for img2



Figure 4: Annotated Coordinates for img3

It should be noted that the four corners of our domain image (“alex_honnold.jpg”) was used. The PQRS regions of our images are included below in tabular format:

Filename	P	Q	R	S
img1.jpg	[424, 856]	[2547, 880]	[2399, 2290]	[684, 3171]
img2.jpg	[524, 1441]	[1866, 820]	[1950, 2810]	[464, 2690]
img3.jpg	[1217, 560]	[2971, 2302]	[1790, 3179]	[248, 1801]
alex_honnold.jpg	[0, 0]	[782, 0]	[782, 664]	[0, 664]

Table 1: P,Q,R,S Regions for our images

We now present our projected domain images:



Figure 5: alex_honnold transformed to img1



Figure 6: alex_honnold transformed to img2

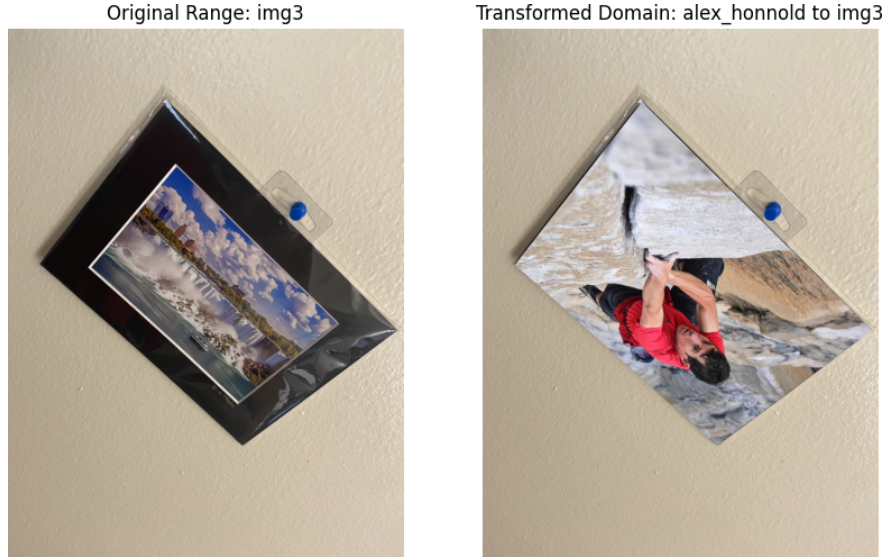


Figure 7: alex_honnold transformed to img3

Task-1.2: Logic

No new logic is particularly needed for this section. However, our domain image is no longer strictly “*alex_honnold.jpg*”. Instead, we will have changing domain and range images to find the homographies between `img1`, `img2`, and `img3`. The homography between `img1` and `img2`, $\mathbf{H}_{ab}$, is found by using domain image points from `img1` and range image points from `img2`. The homography between `img2` and `img3`, $\mathbf{H}_{bc}$, is found by using domain image points from `img2` and range image points from `img3`. We know that multiplying a homography by a homography results in a homography, so we can simply multiply $\mathbf{H}_{ab}$ by $\mathbf{H}_{bc}$ to get our homography from `img1` to `img3`, $\mathbf{H}_{ac}$.

Task-1.2: Implementation

Because we took a modular approach to our python implementation, we simply have to call our `solve_homography` function between our corresponding domain and range images to get the homography matrix between those images. We then utilize `perform_homography` to apply an input homography matrix to a particular image and save the result.

Task-1.2: Results

Our homographies were found to be the following (rounded to the nearest third decimal point). Since we adopted an inverse mapping approach, the homographies listed represent the transformations from the range image back to the domain image. The ‘true’ forward homographies, which map domain image points to range image points, would be the inverses of these matrices:

$$\mathbf{H}_{ab} = \begin{bmatrix} -12.642 & -1.608 & 7970.635 \\ -6.125 & -6.970 & 11292.515 \\ -0.004 & -0.001 & 1.000 \end{bmatrix}, \quad \mathbf{H}_{bc} = \begin{bmatrix} 0.221 & 0.055 & 154.144 \\ -0.510 & 0.034 & 1850.650 \\ 0.000 & 0.000 & 1.000 \end{bmatrix}$$

$$\mathbf{H}_{ac} = \begin{bmatrix} -2.177 & -2.209 & 3045.558 \\ 1.915 & -2.641 & -2550.563 \\ 0.000 & 0.000 & -1.314 \end{bmatrix}$$

And the resulting transformed img1 images are the following:



(a) img1 transformed to img2 range



(b) img1 transformed to img3 range

Task-1.3: Logic

There isn't a significant change between the logic of Task-1.1 and Task-1.3. However, now that we are dealing with strictly affine transformations, we have a slight constraint on our homography matrix, $\mathbf{H}_{\text{affine}}$. Because our homography will be strictly affine, we know that the last row of our matrix will be $[0, 0, 1]$. $\mathbf{H}_{\text{affine}}$ will now be:

$$\mathbf{H}_{\text{affine}} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ 0 & 0 & 1 \end{bmatrix} \quad (11)$$

Which will ultimately affect our system of equations. It should be noted that this expression reflects that 4 correspondence points were used. The reasoning is outlined in the "Task-1.3: Implementation" subsection. For brevity, the step-by-step derivation has been omitted. But we are left with the new representation of our system of equations:

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_2 & y_2 & 1 \\ x_3 & y_3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_3 & y_3 & 1 \\ x_4 & y_4 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_4 & y_4 & 1 \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \end{bmatrix} = \begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \\ x'_4 \\ y'_4 \end{bmatrix} \quad (12)$$

To make this representation more readable, we will simplify Equation 12 to be:

$$\mathbf{A}_{\text{affine}} \mathbf{h}_{\text{affine}} = \mathbf{X}'_{\text{corr}} \quad (13)$$

Since we are interested in the parameters, $\mathbf{h}_{\text{affine}}$, of our affine homography matrix, $\mathbf{H}_{\text{affine}}$, we can see that:

$$\mathbf{h}_{\text{affine}} = \mathbf{A}_{\text{affine}}^{-1} \mathbf{X}'_{\text{corr}} \quad (14)$$

Where we can then reshape $\mathbf{h}_{\text{affine}}$ into our 3x3 affine homography matrix, $\mathbf{H}_{\text{affine}}$; matching its form in Equation 11. Now, to transform our domain points to our range points, we simply follow the operation in Equation 3.

Task-1.3: Implementation

The implementation of the affine transformation in Task-1.3 is similar to that of Task-1.1, with some adjustments to accommodate the constraints given to us by an affine transformation. The main difference lies in the choice of the linear system solver. In Task-1.1, we used `numpy.linalg.solve` to compute the general homography matrix because the system of equations was well-determined with 8 equations and 8 unknowns. However, in Task-1.3, we deal with an affine transformation matrix, which has only 6 unknowns due to the constraint that the last row is fixed to $[0, 0, 1]$.

We initially were using 3 correspondence points that allowed us to use `numpy.linalg.solve` (6 equations for 6 unknowns), but the results were not that favorable. Instead, we wished to use four correspondence points, leading to an overdetermined system (8 equations for 6 unknowns). To solve this overdetermined system, we use `numpy.linalg.lstsq` [4], which is designed to handle overdetermined systems. This function finds a least-squares solution which provides a more stable and accurate affine transformation matrix given we are using more correspondence points than necessary. Of course, we can also continue to use an increasing amount of correspondence points to help increase the quality of our results, but since we used 4 correspondence points in the last section we decided to also use 4 correspondence points for this section.

Once the affine transformation matrix, $\mathbf{H}_{\text{affine}}$, is computed, the transformation is applied similarly to Task-1.1 using the `perform_homography` function. All previous programming considerations laid out in the prior section applies here as well.

Task-1.3: Results

Our mapped images using only affine transformations can be found below:

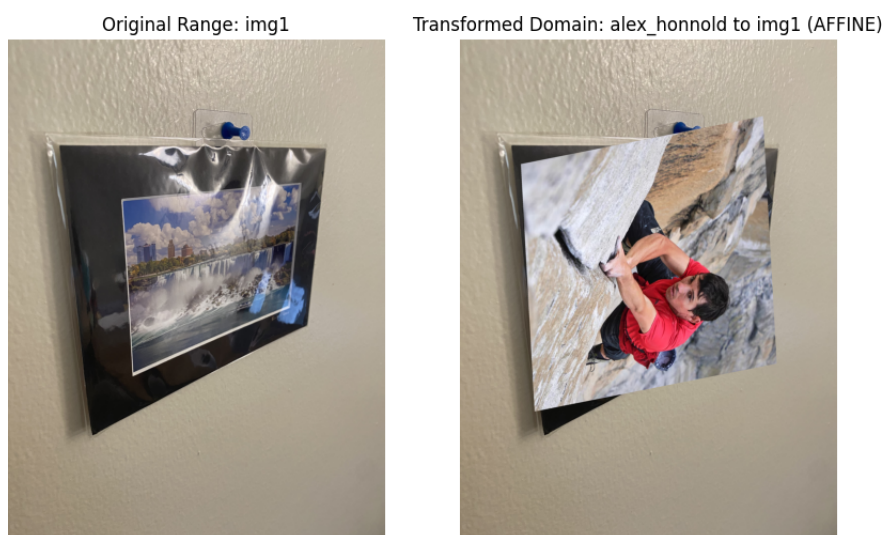


Figure 9: alex_honnold transformed to img1 (affine only)

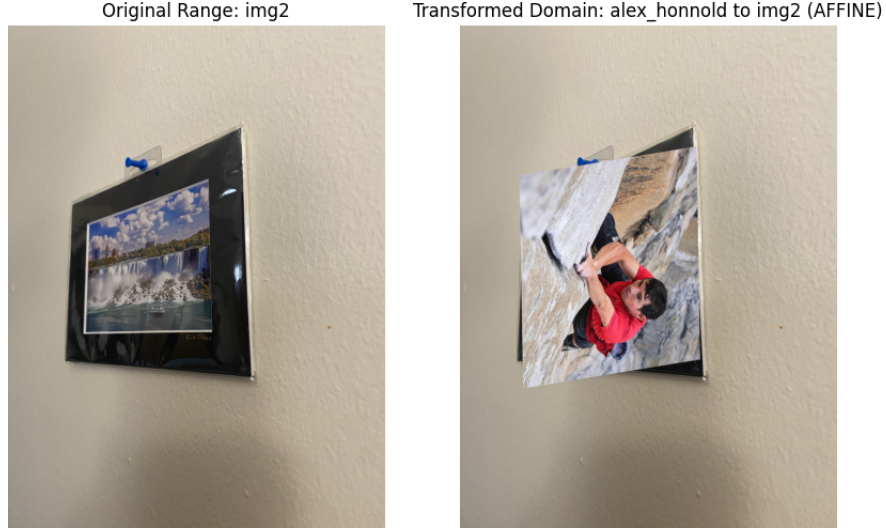


Figure 10: alex_honnold transformed to img2 (affine only)

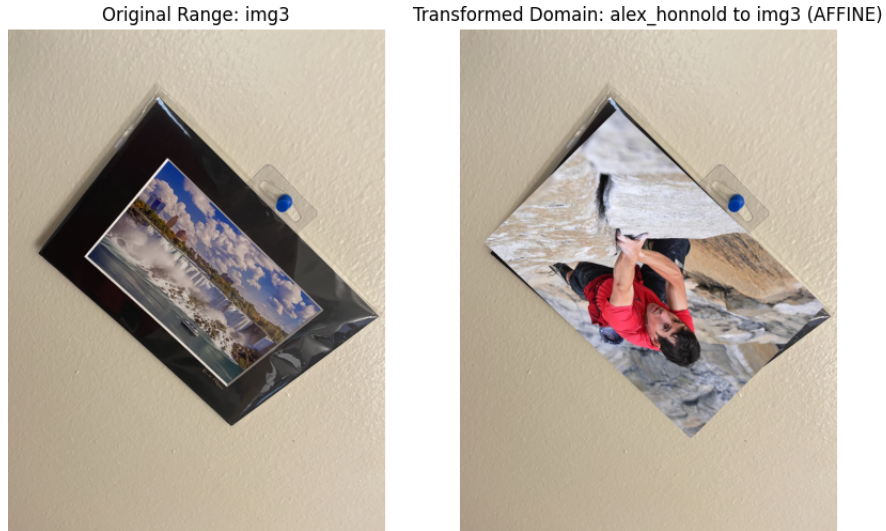


Figure 11: alex_honnold transformed to img3 (affine only)

From Figures 9 to 11, we observe that the results obtained using the affine transformation matrix, $\mathbf{H}_{\text{affine}}$, are less favorable compared to those obtained with the general homography matrix, $\mathbf{H}$. Among the three results, Figure 11 appears to be the most accurate, as it aligns more closely with the PQRS region initially specified in our annotated image.

This difference in performance can be attributed to the properties of affine transformations, which preserve parallel lines. If the PQRS region in our domain image contained parallel lines, these lines would remain parallel after applying an affine transformation. Consequently, the accuracy of the affine transformation depends on the extent to which the PQRS regions in Figures 9 and 10 retained parallelism in their edges. It is possible that these regions were “less parallel” compared to those in Figure 11, leading to a less accurate transformation projection in the former cases.

Task-2

Repeat the steps of Task-1 using your own images. You can capture three images of a planar surface from three different viewpoints such as the ones shown in Figures 1(a-c). For the fourth image you can use your own picture or obtain a picture of an animal, celebrity, etc.

For brevity, it can be assumed that the logic and implementation between Tasks 1 and 2 will be the same. Only results will be shown.

Task-2.1: Results

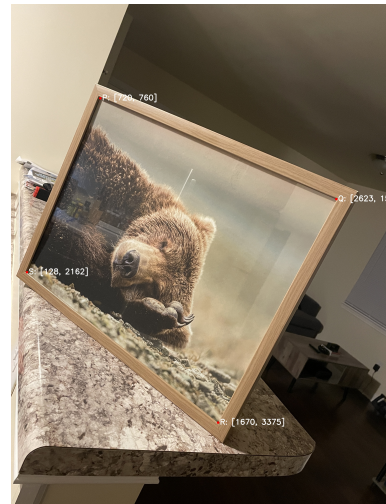
The three pixel coordinate annotated range images which will be used are the following:



(a) bear1 Annotated Coordinates



(b) bear2 Annotated Coordinates



(c) bear3 Annotated Coordinates

Figure 12: Our annotated range images for Task 2

Our domain image will be:



Figure 13: Domain - THE HAGGIS [1]

The pixel coordinates are as follows:

Filename	P	Q	R	S
bear1.jpg	[600, 1045]	[2030, 1185]	[2030, 2346]	[728, 2798]
bear2.jpg	[961, 1417]	[2271, 1321]	[2246, 2730]	[989, 2454]
bear3.jpg	[720, 760]	[2623, 1573]	[1670, 3375]	[128, 2162]
haggis.png	[0, 0]	[1398, 0]	[1398, 693]	[0, 693]

Table 2: P,Q,R,S Regions of our images for Task 2

Our projected images are as follows:



Figure 14: haggis transformed to bear1

Original Range: bear2



Transformed Domain: haggis to bear2



Figure 15: haggis transformed to bear2

Original Range: bear3



Transformed Domain: haggis to bear3



Figure 16: haggis to bear3

Task-2.2: Results

Our homographies were found to be the following (rounded to the nearest third decimal point). Since we adopted an inverse mapping approach, the homographies listed represent the transformations from the range image back to the domain image. The 'true' forward homographies, which map domain image points to range image points, would be the inverses of these matrices:

$$\mathbf{H}_{\text{bear1-bear2}} = \begin{bmatrix} 22.791 & 0.709 & -18093.295 \\ 10.087 & 14.577 & -21963.821 \\ 0.007 & 0.000 & 1.000 \end{bmatrix}, \quad \mathbf{H}_{\text{bear2-bear3}} = \begin{bmatrix} 0.573 & 0.241 & 355.282 \\ -0.291 & 0.558 & 1188.525 \\ 0.000 & -0.000 & 1.000 \end{bmatrix}$$

$$\mathbf{H}_{\text{bear1-bear3}} = \begin{bmatrix} 12.758 & 6.228 & -9152.786 \\ 1.413 & 10.972 & -1055.049 \\ 0.004 & 0.002 & 3.778 \end{bmatrix}$$

And the resulting transformed bear1 images are the following:



(a) bear1 transformed to bear2 range



(b) bear1 transformed to bear3

Task-2.3: Results

Our projected domain images using only affine transformations are below:



Figure 18: haggis transformed to bear1 (affine only)



Figure 19: haggis transformed to bear2 (affine only)

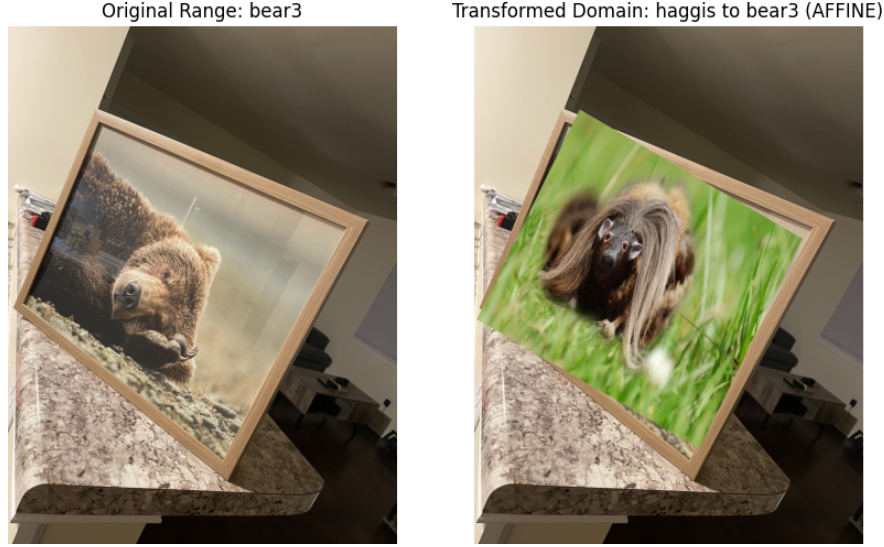


Figure 20: haggis transformed to bear3 (affine only)

From Figures 18-20, we observe that the results obtained using the affine transformation matrix, $\mathbf{H}_{\text{affine}}$, are not particularly favorable across any of the range images. Unlike Task-1, where some PQRS regions contained more parallelism, none of the PQRS regions in the selected range images for Task-2 exhibit notably parallel lines. As a result, the overall accuracy of the transformations appears reduced, and no single result stands out as more accurate than the others.

Extra Credit

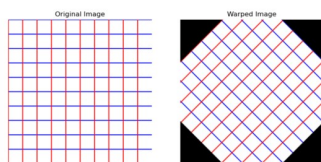
In Lecture 3, we discussed the hierarchy of transformations for homographies and how different homography transformations change the lines and shapes in an image. These concepts are directly applicable to image-editing applications (e.g. **iPhone Photos** and **Snapseed by Google**). These applications are based on the concept of foreground and background, and the editing tools allow you to manipulate the foreground objects in relation to the background. You are also allowed to apply global transformations to an image. For example you could just rotate the entire image. When the foreground objects are manipulated, it is done by applying a homography to the whole image.

Specifically, the **iPhone Photos** application offers the following three operations - (1) **Rotating the image**; (2) **Vertically tilting the image**; and (3) **Horizontally tilting the image**. For Android users, the Snapseed tool does similar things.

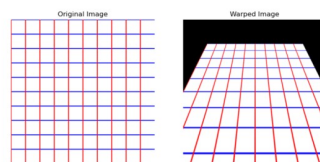
The homographies that are used to produce the effects mentioned above can be parameterized by a single parameter, which represents the degree of change required (e.g. the rotation angle).

Your task for the extra credit is the following:

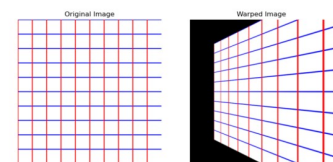
1. **Formulate the parameterized homographies** for each of the three types of operations mentioned above. Representing the parameter by α provide the formula for the corresponding 3x3 homography $\mathbf{H}(\alpha)$.
2. **Apply the homographies** to an input image with horizontal and vertical grid lines and display the resulting images. Your figures should resemble the examples shown in the below figure. Present multiple examples (more than one) for each type of editing operation, along with the corresponding α values.
3. **Analyze the effects** of vertical and horizontal tilting operations on arbitrary vertical and horizontal lines. Discuss which types of lines (vertical or horizontal) remain invariant under each operation. Provide either a proof or numerical evidence to support your conclusions.



(a) Rotation by 45°



(b) Vertical tilting operation



(c) Horizontal tilting operation

In each sub-plot, the left side shows the input image with grid lines, and the right side displays the output image produced by each editing operation.

Extra Credit Problem 1

Rotation Homography Matrix

We know that rotation changes the orientation of an image around a center point. In $\mathbb{R}^2$, this center point will be the physical $\mathbb{R}^2$ origin, $(0, 0)$. Referencing Equation 5, we can think of rotating around the $\mathbb{R}^2$ origin as transforming only the x_1 and x_2 coordinates while we leave the x_3 coordinates unchanged because we are rotating along the x_3 axis. This implies that our homography matrix will resemble the following:

$$\mathbf{H}_{\text{rotation-unknown}}(\alpha) = \begin{bmatrix} h_{11} & h_{12} & 0 \\ h_{21} & h_{22} & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (15)$$

From geometric intuition, we know that an $\mathbb{R}^2$ point, (x_1, x_2) , rotated by an angle, α , has the transformed coordinates (x'_1, x'_2) :

$$x'_1 = x_1 \cos(\alpha) - x_2 \sin(\alpha)$$

$$x'_2 = x_1 \sin(\alpha) + x_2 \cos(\alpha)$$

Keeping in mind Equation 5 and 15, we can reason that the rotation matrix must be the following:

$$\mathbf{H}_{\text{rotation}}(\alpha) = \begin{bmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (16)$$

Vertical Tilting Homography Matrix

Vertical tilting will skew the image vertically, implying that we keep the x_1 coordinates unchanged while transforming the x_2 and x_3 coordinates because we are rotating along the x_1 axis. To reflect this notion, we have the following expressions:

$$x'_1 = x_1$$

$$x'_2 = x_2 \cos(\alpha) + x_3 \sin(\alpha)$$

$$x'_3 = -x_2 \sin(\alpha) + x_3 \cos(\alpha)$$

In HC representation, $x_3 = 1$, so our vertical tilt matrix will be the following:

$$\mathbf{H}_{\text{vertical-tilt}}(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & \sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{bmatrix} \quad (17)$$

Horizontal Tilting Homography Matrix

Horizontal tilting will skew the image horizontally, implying that we keep the x_2 coordinates unchanged while transforming the x_1 and x_3 coordinates because we are rotating along the x_2 axis. To reflect this notion, we have the following expressions:

$$x'_1 = x_1 \cos(\alpha) - x_3 \sin(\alpha)$$

$$x'_2 = x_2$$

$$x'_3 = x_1 \sin(\alpha) + x_3 \cos(\alpha)$$

In HC representation, $x_3 = 1$, so our horizontal tilt matrix will be the following:

$$\mathbf{H}_{\text{horizontal-tilt}}(\alpha) = \begin{bmatrix} \cos(\alpha) & 0 & -\sin(\alpha) \\ 0 & 1 & 0 \\ \sin(\alpha) & 0 & \cos(\alpha) \end{bmatrix} \quad (18)$$

Extra Credit Problem 2

The following gridline image was generated using Python and was used for our transformations. An “arrow-like” structure was included to help assess orientation:

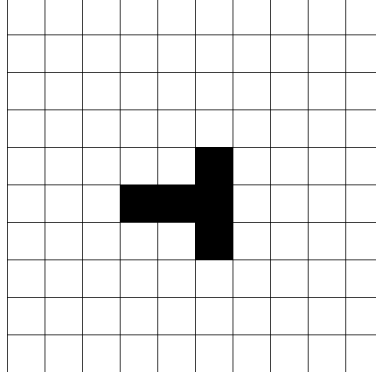


Figure 21: Original gridline image (generated by Python)

The gridline image was transformed with rotation, vertical tilting, and horizontal tilting for $+15$, $+45$, and -45 degrees. The results are found below:

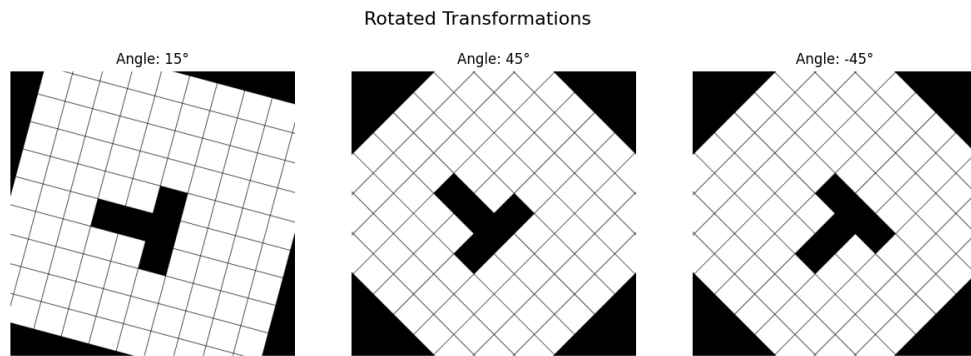


Figure 22: The original gridline image rotated by our predefined angles

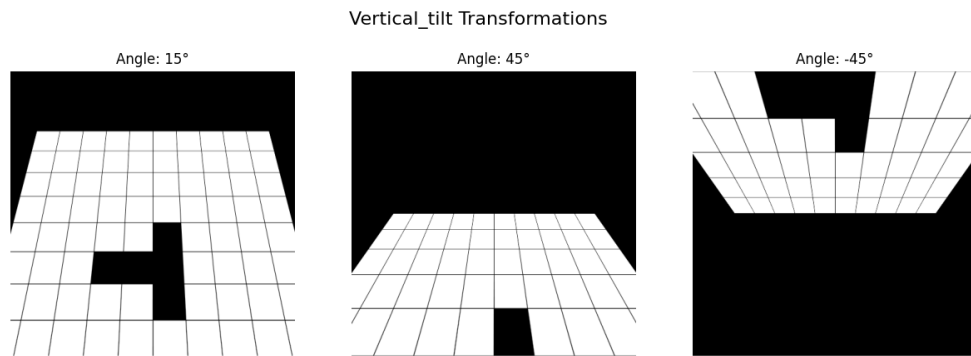


Figure 23: The original gridline image vertically tilted by our predefined angles

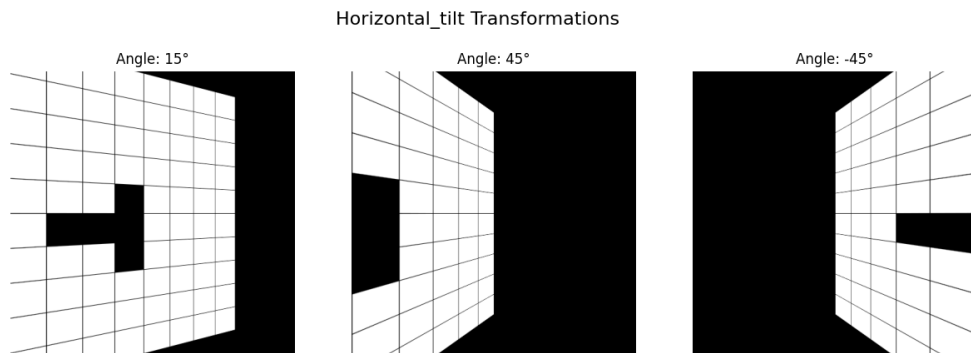


Figure 24: The original gridline image horizontally tilted by our predefined angles

Extra Credit Problem 3

As previously mentioned, when applying a vertical or horizontal tilting transformation to an image, we rotate about a particular axis, which affects lines differently depending on their alignment to that axis. For vertical tilting, we rotate about the x_1 axis which will affect horizontal lines, making them appear slanted while keeping vertical lines unchanged. Conversely, for horizontal tilting, we rotate about the x_2 axis which will affect vertical lines, making them appear slanted while keeping horizontal lines unchanged.

To numerically prove this, we will utilize two separate lines, one parallel with the x-axis and the other parallel with the y-axis, apply both vertical and horizontal tilting, observe the results. If a skew becomes present, then a x_1 or x_2 component will be added that was not there before. $\mathbf{l}_1$ is parallel to the x-axis, and $\mathbf{l}_2$ is parallel to the y-axis:

$$\mathbf{l}_1 = \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix}, \quad \mathbf{l}_2 = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}$$

We also have to keep in mind how to properly apply homography matrices to lines to arrive at their transformed counterparts:

$$\mathbf{l}' = \mathbf{H}^{-T} \mathbf{l} \quad (19)$$

Now we will apply a horizontal tilt of +15 degrees to both of these lines:

$$\mathbf{l}_{1\text{-horz}} = \mathbf{H}_{\text{horizontal-tilt}}(15^\circ)^{-T} \mathbf{l}_1 = \begin{bmatrix} \cos(15) & 0 & -\sin(15) \\ 0 & 1 & 0 \\ \sin(15) & 0 & \cos(15) \end{bmatrix}^{-T} \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 0.258 \\ 1 \\ -0.966 \end{bmatrix}$$

$$\mathbf{l}_{2\text{-horz}} = \mathbf{H}_{\text{horizontal-tilt}}(15^\circ)^{-T} \mathbf{l}_2 = \begin{bmatrix} \cos(15) & 0 & -\sin(15) \\ 0 & 1 & 0 \\ \sin(15) & 0 & \cos(15) \end{bmatrix}^{-T} \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} = \begin{bmatrix} 1.225 \\ 0 \\ -0.707 \end{bmatrix}$$

As we thought, $\mathbf{l}_1$ receives a skew in the form of an added x_1 component while $\mathbf{l}_2$ does not receive any additional component. Now we will apply a vertical tilt of +15 degrees to both of these lines:

$$\mathbf{l}_{1\text{-vert}} = \mathbf{H}_{\text{vertical-tilt}}(15^\circ)^{-T} \mathbf{l}_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(15) & \sin(15) \\ 0 & \sin(15) & \cos(15) \end{bmatrix}^{-T} \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0.707 \\ -1.225 \end{bmatrix}$$

$$\mathbf{l}_{2\text{-vert}} = \mathbf{H}_{\text{vertical-tilt}}(15^\circ)^{-T} \mathbf{l}_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(15) & \sin(15) \\ 0 & \sin(15) & \cos(15) \end{bmatrix}^{-T} \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} = \begin{bmatrix} 1 \\ -0.258 \\ -0.966 \end{bmatrix}$$

Similarly, a skew presented itself in $\mathbf{l}_2$ in the form of an added x_1 component while $\mathbf{l}_1$ did not receive any additional component.

Python Code: Toplevel Script

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 import cv2
4 import os
5 import json
6
7 from master_helper import print_threeXthree_array,
  print_with_delimiter
8
9
10 def solve_affine(X_prime, X):
11     """
12     Function to solve the Affine transformation matrix A based on
13     dictionaries X_prime and X.
14
15     Inputs:
16         X_prime (dict): Dictionary of destination points with keys
17         'x' and 'y'.
18         X (dict): Dictionary of source points with keys 'x' and 'y'.
19
20     Returns the affine transformation matrix, H_affine.
21     """
22     # Initialize the matrix A
23     A = []
24
25     # Converting the dictionary values to NumPy arrays
26     x = np.array(X["x"]) # Domain x points
27     y = np.array(X["y"]) # Domain y points
28     x_prime = np.array(X_prime["x"]) # Range x' points
29     y_prime = np.array(X_prime["y"]) # Range y' points
30
31     # Loop through each pair of points and construct the rows of
32     unknowns matrix, A
33     for i in range(4): # We have 4 correspondence points
34         # Construct the two rows for the ith point
35         A.append([x[i], y[i], 1, 0, 0, 0])
36         A.append([0, 0, 0, x[i], y[i], 1])
37
38     # Convert A to a NumPy array
39     A = np.array(A)
40
41     # Create vector chi_prime from X_prime coordinates
42     chi_prime = np.array([
43         x_prime[0],
```

```

42         y_prime[0],
43         x_prime[1],
44         y_prime[1],
45         x_prime[2],
46         y_prime[2],
47         x_prime[3],
48         y_prime[3],
49     ]
50 )
51
52 # Solve the system of linear equations to find the affine
53 # parameters
54 # Note: Because we have an over-determined system (8 equations,
55 # 6 unknowns), we need to use linalg.lstsq
56 H_params = np.linalg.lstsq(A, chi_prime, rcond=None)[0]
57
58 # Append [0, 0, 1] to the parameter array to align with an
59 # affine matrix
60 H_params = np.append(H_params, [0, 0, 1])
61
62 # Reshape the parameter array into the affine matrix H_affine
63 H_affine = H_params.reshape((3, 3))
64
65 return H_affine
66
67 def solve_homography(X_prime, X):
68     """
69     Function to solve the Homography matrix H based on dictionaries
70     X_prime and X.
71
72     Inputs:
73     X_prime (dict): Dictionary of destination points with keys
74     'x' and 'y'.
75     X (dict): Dictionary of source points with keys 'x' and 'y'.
76
77     Returns general homography matrix, H.
78     """
79     # Initialize the matrix A
80     A = []
81
82     # Converting the dictionary values to NumPy arrays
83     x = np.array(X["x"]) # Domain x points
84     y = np.array(X["y"]) # Domain y points
85     x_prime = np.array(X_prime["x"]) # Range x' points
86     y_prime = np.array(X_prime["y"]) # Range y' points

```

```

84
85 # Loop through each pair of points and construct the rows of A
86 for i in range(4): # Four correspondence points
87     # Construct the two rows for the ith point
88     A.append([x[i], y[i], 1, 0, 0, 0, -x[i] * x_prime[i], -y[i]
89               * x_prime[i]])
90     A.append([0, 0, 0, x[i], y[i], 1, -x[i] * y_prime[i], -y[i]
91               * y_prime[i]])
92
93 # Convert A to a NumPy array
94 A = np.array(A)
95
96 # Create vector chi_prime from X_prime coordinates
97 chi_prime = np.array(
98     [
99         x_prime[0],
100        y_prime[0],
101        x_prime[1],
102        y_prime[1],
103        x_prime[2],
104        y_prime[2],
105        x_prime[3],
106        y_prime[3],
107    ]
108 )
109
110 # Solve the system of linear equations to find the homography
111 # parameters
112 # Note: We can use linalg.solve because we have a
113 # well-determined system (8 equations, 8 unknowns)
114 H_params = np.linalg.solve(A, chi_prime)
115
116 # Append 1 to the parameter array to complete the homography
117 # matrix
118 H_params = np.append(H_params, 1)
119
120 # Reshape the parameter array into the homography matrix H
121 H = H_params.reshape((3, 3))
122
123 return H
124
125 def solve_and_perform_homography(domain_img_data, range_img_data):
126     """
127     Function to apply homography transformation from domain to
128     range image data.

```

```

125     Inputs:
126         domain_img_data (dict): Dictionary containing domain image
127         range_img_data (dict): Dictionary containing range image
128         file path and points.
129         range_img_data (dict): Dictionary containing range image
130         file path and points.
131
132     Returns tuple of original and transformed range images and
133     homography matrix.
134     """
135
136     # Load images using the file paths from the dictionaries
137     range_img = cv2.imread(range_img_data["filepath"])
138     range_img_orig = np.copy(range_img)
139     domain_img = cv2.imread(domain_img_data["filepath"])
140
141     # Solve the homography matrix using the coordinates, not images
142     H = solve_homography(domain_img_data, range_img_data)
143
144     # Perform homography transformation using the solved matrix
145     transformed_img = perform_homography(domain_img, range_img, H)
146
147     return range_img_orig, transformed_img, H
148
149 def solve_and_perform_homography_affine(domain_img_data,
150 range_img_data):
151     """
152     Function to apply homography transformation from domain to
153     range image data.
154
155     Inputs:
156         domain_img_data (dict): Dictionary containing domain image
157         range_img_data (dict): Dictionary containing range image
158         file path and points.
159         range_img_data (dict): Dictionary containing range image
160         file path and points.
161
162     Returns tuple of original and transformed range images and
163     homography matrix.
164     """
165
166     # Load images using the file paths from the dictionaries
167     range_img = cv2.imread(range_img_data["filepath"])
168     range_img_orig = np.copy(range_img)
169     domain_img = cv2.imread(domain_img_data["filepath"])

```

```

164 # Solve the homography matrix using the coordinates, not images
165 H_affine = solve_affine(domain_img_data, range_img_data)
166
167 # Perform homography transformation using the solved matrix
168 transformed_img = perform_homography(domain_img, range_img,
169                                     H_affine)
170
171 return range_img_orig, transformed_img, H_affine
172
173 def perform_homography(domain_img, range_img, H):
174     """
175     Function to apply homography transformation from domain to
176     range image using a given homography matrix.
177
178     Inputs:
179         domain_img (np array): Domain image.
180         range_img (np array): Range image to transform.
181         H (np array): Homography matrix.
182
183     Returns transformed range image (np array).
184     """
185
186     # Get dimensions of the range image
187     height, width = range_img.shape[:2]
188
189     # Generate a grid of (x, y) coordinates for every pixel in the
190     # range image
191     x_coords, y_coords = np.meshgrid(np.arange(width),
192                                     np.arange(height))
193
194     # Flatten the coordinate grids to create a list of coordinates
195     x_coords_flat = x_coords.flatten()
196     y_coords_flat = y_coords.flatten()
197
198     # Create an array of homogeneous coordinates for the points
199     ones_HC = np.ones_like(x_coords_flat)
200     points_HC = np.vstack((x_coords_flat, y_coords_flat, ones_HC))
201     # Shape: (3, N)
202
203     # Apply the homography transformation using matrix
204     # multiplication
205     transformed_points = (
206         H @ points_HC
207     ) # Matrix multiplication with H (3x3) and points_HC (3xN)
208
209     # Normalize by the third coordinate to convert back to HC

```

```

205     transformed_points /= transformed_points[2, :]
206
207     # Round and convert to integer for pixel coordinates
208     transformed_points = np rint(transformed_points).astype(int)
209
210     # Filter out points that are outside the target image bounds
211     valid_indices = (
212         (transformed_points[0, :] >= 0)
213         & (transformed_points[0, :] < domain_img.shape[1])
214         & (transformed_points[1, :] >= 0)
215         & (transformed_points[1, :] < domain_img.shape[0])
216     )
217
218     # Get valid transformed coordinates
219     valid_transformed_points = transformed_points[:, valid_indices]
220     valid_original_points = points_HC[:, valid_indices]
221
222     # Assign pixel values from the source image to the target image
223     range_img[valid_original_points[1], valid_original_points[0]] =
224         domain_img[
225             valid_transformed_points[1], valid_transformed_points[0]
226         ]
227
228     return range_img
229
230 def task_1(img_data, show_plots):
231     """
232     Function to perform the described subtasks from Task-1
233     """
234
235     # Extract image points and file paths from the loaded JSON data
236     alex_img = img_data["alex"]
237     img1 = img_data["img1"]
238     img2 = img_data["img2"]
239     img3 = img_data["img3"]
240
241     """
242     Task 1.1: Project Figure 1(d) onto Figures 1(a-c) PQRS region
243         using homographies
244     """
245
246     # Perform homography transformation
247     range_img1, range_img1_alex, H_da =
248         solve_and_perform_homography(alex_img, img1)
249     range_img2, range_img2_alex, H_db =
250         solve_and_perform_homography(alex_img, img2)

```



```

248 range_img3, range_img3_alex, H_dc =
    solve_and_perform_homography(alex_img, img3)
249
250 # Print 3x3 Homographies (for logging)
251 print_with_delimiter("H_da")
252 print_threeXthree_array(H_da)
253
254 print_with_delimiter("H_db")
255 print_threeXthree_array(H_db)
256
257 print_with_delimiter("H_dc")
258 print_threeXthree_array(H_dc)
259
260 # =====
261 # --- PLOTTING
262 # =====
263 # Display the results of Domain: alex_honnold.jpg, Range:
    img1.jpg
264 plt.figure(figsize=(10, 10))
265 plt.subplot(121)
266 plt.title("Original Range: img1")
267 plt.imshow(cv2.cvtColor(range_img1, cv2.COLOR_BGR2RGB))
268 plt.axis("off")
269 plt.grid(False)
270 plt.subplot(122)
271 plt.title("Transformed Domain: alex_honnold to img1")
272 plt.imshow(cv2.cvtColor(range_img1_alex, cv2.COLOR_BGR2RGB))
273 plt.axis("off")
274 plt.grid(False)
275 plt.savefig(os.path.join(figs_dir, "alex_to_img1.png"))
276 if show_plots:
277     plt.show()
278
279 # Display the results of Domain: alex_honnold.jpg, Range:
    img2.jpg
280 plt.figure(figsize=(10, 10))
281 plt.subplot(121)
282 plt.title("Original Range: img2")
283 plt.imshow(cv2.cvtColor(range_img2, cv2.COLOR_BGR2RGB))
284 plt.axis("off")
285 plt.grid(False)
286 plt.subplot(122)
287 plt.title("Transformed Domain: alex_honnold to img2")
288 plt.imshow(cv2.cvtColor(range_img2_alex, cv2.COLOR_BGR2RGB))
289 plt.axis("off")
290 plt.grid(False)
291 plt.savefig(os.path.join(figs_dir, "alex_to_img2.png"))

```

```

292     if show_plots:
293         plt.show()
294
295     # Display the results of Domain: alex_honnold.jpg, Range:
296         img3.jpg
297     plt.figure(figsize=(10, 10))
298     plt.subplot(121)
299     plt.title("Original Range: img3")
300     plt.imshow(cv2.cvtColor(range_img3, cv2.COLOR_BGR2RGB))
301     plt.axis("off")
302     plt.grid(False)
303     plt.subplot(122)
304     plt.title("Transformed Domain: alex_honnold to img3")
305     plt.imshow(cv2.cvtColor(range_img3_alex, cv2.COLOR_BGR2RGB))
306     plt.axis("off")
307     plt.grid(False)
308     plt.savefig(os.path.join(figs_dir, "alex_to_img3.png"))
309     if show_plots:
310         plt.show()
311
312     """
313     Task 1.2: Find homographies between Figures 1(a-b), 1(b-c)
314     """
315
316     # Calculate the homography between img1 and img2, then between
317         img2 and img3
318     H_ab = solve_homography(img1, img2)
319     H_bc = solve_homography(img2, img3)
320
321     H_ac = H_ab @ H_bc
322
323     print_with_delimiter("H_ab")
324     print_threeXthree_array(H_ab)
325
326     print_with_delimiter("H_bc")
327     print_threeXthree_array(H_bc)
328
329     print_with_delimiter("H_ac")
330     print_threeXthree_array(H_ac)
331
332     # Apply homographies H_ab and H_bc to img1
333     img1_to_img2 = perform_homography(
334         cv2.imread(img1["filepath"]), cv2.imread(img2["filepath"]),
335         H_ab
336     )
337     img1_to_img3 = perform_homography(img1_to_img2,
338         cv2.imread(img3["filepath"]), H_bc)

```

```

335 # =====
336 # --- PLOTTING
337 # =====
338
339
340 # Display the results of Domain: img1.jpg, Range: img2.jpg
341 plt.figure(figsize=(10, 10))
342 plt.title("Transformed img1 to img2 using H_ab")
343 plt.imshow(cv2.cvtColor(img1_to_img2, cv2.COLOR_BGR2RGB))
344 plt.axis("off")
345 plt.grid(False)
346 plt.savefig(os.path.join(figs_dir, "img1_to_img2.png"))
347 if show_plots:
348     plt.show()
349
350 # Display the results of Domain: TRANSFORMED img1.jpg, Range:
    img3.jpg
351 plt.figure(figsize=(10, 10))
352 plt.title("Transformed img1 to img3 using H_ab and H_bc")
353 plt.imshow(cv2.cvtColor(img1_to_img3, cv2.COLOR_BGR2RGB))
354 plt.axis("off")
355 plt.grid(False)
356 plt.savefig(os.path.join(figs_dir, "img1_to_img3.png"))
357 if show_plots:
358     plt.show()
359
360 """
361 Task 1.3: Project Figure 1(d) onto Figures 1(a-c) PQRS region
    using only affine homographies
362 """
363
364 # Perform homography transformation (affine)
365 range_img1, range_img1_alex_affine, H_da_affine = (
366     solve_and_perform_homography_affine(alex_img, img1)
367 )
368 range_img2, range_img2_alex_affine, H_db_affine = (
369     solve_and_perform_homography_affine(alex_img, img2)
370 )
371 range_img3, range_img3_alex_affine, H_dc_affine = (
372     solve_and_perform_homography_affine(alex_img, img3)
373 )
374
375 # Print 3x3 Homographies (for logging)
376 print_with_delimiter("H_da_affine")
377 print_threeXthree_array(H_da_affine)
378
379 print_with_delimiter("H_db_affine")

```

```

380 print_threeXthree_array(H_db_affine)
381
382 print_with_delimiter("H_dc_affine")
383 print_threeXthree_array(H_dc_affine)
384
385 # =====
386 # --- PLOTTING
387 # =====
388 # Display the results of Domain: alex_honnold.jpg, Range:
    img1.jpg (AFFINE)
389 plt.figure(figsize=(10, 10))
390 plt.subplot(121)
391 plt.title("Original Range: img1")
392 plt.imshow(cv2.cvtColor(range_img1, cv2.COLOR_BGR2RGB))
393 plt.axis("off")
394 plt.grid(False)
395 plt.subplot(122)
396 plt.title("Transformed Domain: alex_honnold to img1 (AFFINE)")
397 plt.imshow(cv2.cvtColor(range_img1_alex_affine,
    cv2.COLOR_BGR2RGB))
398 plt.axis("off")
399 plt.grid(False)
400 plt.savefig(os.path.join(figs_dir, "alex_to_img1_affine.png"))
401 if show_plots:
402     plt.show()
403
404 # Display the results of Domain: alex_honnold.jpg, Range:
    img2.jpg (AFFINE)
405 plt.figure(figsize=(10, 10))
406 plt.subplot(121)
407 plt.title("Original Range: img2")
408 plt.imshow(cv2.cvtColor(range_img2, cv2.COLOR_BGR2RGB))
409 plt.axis("off")
410 plt.grid(False)
411 plt.subplot(122)
412 plt.title("Transformed Domain: alex_honnold to img2 (AFFINE)")
413 plt.imshow(cv2.cvtColor(range_img2_alex_affine,
    cv2.COLOR_BGR2RGB))
414 plt.axis("off")
415 plt.grid(False)
416 plt.savefig(os.path.join(figs_dir, "alex_to_img2_affine.png"))
417 if show_plots:
418     plt.show()
419
420 # Display the results of Domain: alex_honnold.jpg, Range:
    img3.jpg (AFFINE)
421 plt.figure(figsize=(10, 10))

```

```

422 plt.subplot(121)
423 plt.title("Original Range: img3")
424 plt.imshow(cv2.cvtColor(range_img3, cv2.COLOR_BGR2RGB))
425 plt.axis("off")
426 plt.grid(False)
427 plt.subplot(122)
428 plt.title("Transformed Domain: alex_honnold to img3 (AFFINE)")
429 plt.imshow(cv2.cvtColor(range_img3_alex_affine,
430                     cv2.COLOR_BGR2RGB))
431 plt.axis("off")
432 plt.grid(False)
433 plt.savefig(os.path.join(figs_dir, "alex_to_img3_affine.png"))
434 if show_plots:
435     plt.show()
436
437 def task_2(img_data, show_plots):
438     """
439     Function to perform the described subtasks from Task-2
440     """
441     # Extract image points and file paths from the loaded JSON data
442     haggis = img_data["haggis"]
443     bear1 = img_data["bear1"]
444     bear2 = img_data["bear2"]
445     bear3 = img_data["bear3"]
446
447     """
448     Task 2.1: Project haggis.png onto our 3 bear image PQRS regions
449     using homographies
450     """
451
452     # Perform homography transformation
453     range_bear1, range_bear1_haggis, H_haggis_bear1 =
454         solve_and_perform_homography(
455             haggis, bear1
456         )
457     range_bear2, range_bear2_haggis, H_haggis_bear2 =
458         solve_and_perform_homography(
459             haggis, bear2
460         )
461     range_bear3, range_bear3_haggis, H_haggis_bear3 =
462         solve_and_perform_homography(
463             haggis, bear3
464         )
465
466     # Print 3x3 Homographies (for logging)
467     print_with_delimiter("H_haggis_bear1")

```

```

464 print_threeXthree_array(H_haggis_bear1)
465
466 print_with_delimiter("H_haggis_bear2")
467 print_threeXthree_array(H_haggis_bear2)
468
469 print_with_delimiter("H_haggis_bear3")
470 print_threeXthree_array(H_haggis_bear3)
471
472 # =====
473 # --- PLOTTING
474 # =====
475 # Display the results of Domain: haggis.png, Range: bear1.jpg
476 plt.figure(figsize=(10, 10))
477 plt.subplot(121)
478 plt.title("Original Range: bear1")
479 plt.imshow(cv2.cvtColor(range_bear1, cv2.COLOR_BGR2RGB))
480 plt.axis("off")
481 plt.grid(False)
482 plt.subplot(122)
483 plt.title("Transformed Domain: haggis to bear1")
484 plt.imshow(cv2.cvtColor(range_bear1_haggis, cv2.COLOR_BGR2RGB))
485 plt.axis("off")
486 plt.grid(False)
487 plt.savefig(os.path.join(figs_dir, "haggis_to_bear1.png"))
488 if show_plots:
489     plt.show()
490
491 # Display the results of Domain: haggis.png, Range: bear2.jpg
492 plt.figure(figsize=(10, 10))
493 plt.subplot(121)
494 plt.title("Original Range: bear2")
495 plt.imshow(cv2.cvtColor(range_bear2, cv2.COLOR_BGR2RGB))
496 plt.axis("off")
497 plt.grid(False)
498 plt.subplot(122)
499 plt.title("Transformed Domain: haggis to bear2")
500 plt.imshow(cv2.cvtColor(range_bear2_haggis, cv2.COLOR_BGR2RGB))
501 plt.axis("off")
502 plt.grid(False)
503 plt.savefig(os.path.join(figs_dir, "haggis_to_bear2.png"))
504 if show_plots:
505     plt.show()
506
507 # Display the results of Domain: haggis.png, Range: bear3.jpg
508 plt.figure(figsize=(10, 10))
509 plt.subplot(121)
510 plt.title("Original Range: bear3")

```

```

511 plt.imshow(cv2.cvtColor(range_bear3, cv2.COLOR_BGR2RGB))
512 plt.axis("off")
513 plt.grid(False)
514 plt.subplot(122)
515 plt.title("Transformed Domain: haggis to bear3")
516 plt.imshow(cv2.cvtColor(range_bear3_haggis, cv2.COLOR_BGR2RGB))
517 plt.axis("off")
518 plt.grid(False)
519 plt.savefig(os.path.join(figs_dir, "haggis_to_bear3.png"))
520 if show_plots:
521     plt.show()
522
523 """
524 Task 1.2: Find homographies between bear1-bear2, bear2-bear3
525 """
526
527 # Calculate the homography between bear1 and bear2, then
    between bear2 and bear3
528 H_bear1_bear2 = solve_homography(bear1, bear2)
529 H_bear2_bear3 = solve_homography(bear2, bear3)
530
531 H_bear1_bear3 = H_bear1_bear2 @ H_bear2_bear3
532
533 print_with_delimiter("H_bear1_bear2")
534 print_threeXthree_array(H_bear1_bear2)
535
536 print_with_delimiter("H_bear2_bear3")
537 print_threeXthree_array(H_bear2_bear3)
538
539 print_with_delimiter("H_bear1_bear3")
540 print_threeXthree_array(H_bear1_bear3)
541
542 # Apply homographies H_bear1_bear2 and H_bear2_bear3 to bear1
543 bear1_to_bear2 = perform_homography(
544     cv2.imread(bear1["filepath"]),
        cv2.imread(bear2["filepath"]), H_bear1_bear2
545 )
546 bear1_to_bear3 = perform_homography(
547     bear1_to_bear2, cv2.imread(bear3["filepath"]), H_bear2_bear3
548 )
549
550 # =====
551 # --- PLOTTING
552 # =====
553 # Display the results of Domain: bear1.jpg, Range: bear2.jpg
554 plt.figure(figsize=(10, 10))
555 plt.title("Transformed bear1 to bear2 using H_bear1_bear2")

```



```

556 plt.imshow(cv2.cvtColor(bear1_to_bear2, cv2.COLOR_BGR2RGB))
557 plt.axis("off")
558 plt.grid(False)
559 plt.savefig(os.path.join(figs_dir, "bear1_to_bear2.png"))
560 if show_plots:
561     plt.show()
562
563 # Display the results of Domain: TRANSFORMED bear1.jpg, Range:
    bear3.jpg
564 plt.figure(figsize=(10, 10))
565 plt.title("Transformed bear1 to bear3 using H_bear1_bear2 and
    H_bear2_bear3")
566 plt.imshow(cv2.cvtColor(bear1_to_bear3, cv2.COLOR_BGR2RGB))
567 plt.axis("off")
568 plt.grid(False)
569 plt.savefig(os.path.join(figs_dir, "bear1_to_bear3.png"))
570 if show_plots:
571     plt.show()
572
573 """
574 Task 1.3: Project haggis.png onto our 3 bear image PQRS regions
    using only affine homographies
575 """
576
577 # Perform homography transformation (affine)
578 range_bear1, range_bear1_haggis_affine, H_haggis_bear1_affine =
    (
579     solve_and_perform_homography_affine(haggis, bear1)
580 )
581 range_bear2, range_bear2_haggis_affine, H_haggis_bear2_affine =
    (
582     solve_and_perform_homography_affine(haggis, bear2)
583 )
584 range_bear3, range_bear3_haggis_affine, H_haggis_bear3_affine =
    (
585     solve_and_perform_homography_affine(haggis, bear3)
586 )
587
588 # Print 3x3 Homographies (for logging)
589 print_with_delimiter("H_haggis_bear1_affine")
590 print_threeXthree_array(H_haggis_bear1_affine)
591
592 print_with_delimiter("H_haggis_bear2_affine")
593 print_threeXthree_array(H_haggis_bear2_affine)
594
595 print_with_delimiter("H_haggis_bear3_affine")
596 print_threeXthree_array(H_haggis_bear3_affine)

```

```

597
598 # =====
599 # --- PLOTTING
600 # =====
601 # Display the results of Domain: haggis.png, Range: bear1.jpg
    (AFFINE)
602 plt.figure(figsize=(10, 10))
603 plt.subplot(121)
604 plt.title("Original Range: bear1")
605 plt.imshow(cv2.cvtColor(range_bear1, cv2.COLOR_BGR2RGB))
606 plt.axis("off")
607 plt.grid(False)
608 plt.subplot(122)
609 plt.title("Transformed Domain: haggis to bear1 (AFFINE)")
610 plt.imshow(cv2.cvtColor(range_bear1_haggis_affine,
    cv2.COLOR_BGR2RGB))
611 plt.axis("off")
612 plt.grid(False)
613 plt.savefig(os.path.join(figs_dir,
    "haggis_to_bear1_affine.png"))
614 if show_plots:
615     plt.show()
616
617 # Display the results of Domain: haggis.png, Range: bear2.jpg
    (AFFINE)
618 plt.figure(figsize=(10, 10))
619 plt.subplot(121)
620 plt.title("Original Range: bear2")
621 plt.imshow(cv2.cvtColor(range_bear2, cv2.COLOR_BGR2RGB))
622 plt.axis("off")
623 plt.grid(False)
624 plt.subplot(122)
625 plt.title("Transformed Domain: haggis to bear2 (AFFINE)")
626 plt.imshow(cv2.cvtColor(range_bear2_haggis_affine,
    cv2.COLOR_BGR2RGB))
627 plt.axis("off")
628 plt.grid(False)
629 plt.savefig(os.path.join(figs_dir,
    "haggis_to_bear2_affine.png"))
630 if show_plots:
631     plt.show()
632
633 # Display the results of Domain: haggis.png, Range: bear3.jpg
    (AFFINE)
634 plt.figure(figsize=(10, 10))
635 plt.subplot(121)
636 plt.title("Original Range: bear3")

```

```

637 plt.imshow(cv2.cvtColor(range_bear3, cv2.COLOR_BGR2RGB))
638 plt.axis("off")
639 plt.grid(False)
640 plt.subplot(122)
641 plt.title("Transformed Domain: haggis to bear3 (AFFINE)")
642 plt.imshow(cv2.cvtColor(range_bear3_haggis_affine,
643                        cv2.COLOR_BGR2RGB))
644 plt.axis("off")
645 plt.grid(False)
646 plt.savefig(os.path.join(figs_dir,
647                        "haggis_to_bear3_affine.png"))
648 if show_plots:
649     plt.show()
650
651 if __name__ == "__main__":
652
653     # Boolean to display plots (MAINLY FOR DEBUG)
654     show_plots_bool = False
655
656     # Load the image data from the JSON file
657     json_file = "img_data.json"
658     with open(json_file, "r") as f:
659         img_data = json.load(f)
660
661     # Directory to save figures
662     figs_dir = "HW2_images"
663     os.makedirs(figs_dir, exist_ok=True)
664
665     # *****
666     # TASK 1: img1, img2, img3, and Alex Honnold
667     # *****
668     task_1(img_data, show_plots=show_plots_bool)
669
670     # *****
671     # TASK 2: bear1, bear2, bear3, and THE HAGGIS
672     # *****
673     task_2(img_data, show_plots=show_plots_bool)

```

Python Code: Script to retrieve pixel coordinates

```

1 """
2 This online post:
3     https://www.geeksforgeeks.org/displaying-the-coordinates-of-the-points-click
    served as the GUI foundation for clicking our PQRS region. The
    provided code was heavily modified to cater more to our needs.

```

```

4  """
5
6  import cv2
7  import sys
8  import json
9  import os
10 from master_helper import print_with_delimiter
11 from master_helper import Logger
12
13 # Global variables to store the image and click count
14 img = None
15 click_count = 0
16 letters = ["P", "Q", "R", "S"]
17 coordinates = {"x": [], "y": []} # To store the clicked coordinates
18
19
20 def print_image_corners(image_filepath):
21     """
22     Extracts and prints the pixel coordinates for the four corners
23     of an image.
24     Also returns the coordinates for JSON storage.
25
26     :param image_filepath: The path to the image file.
27     :return: Dictionary containing the corner coordinates.
28     """
29     # Read in the image
30     img = cv2.imread(image_filepath)
31
32     # Check if the image is loaded correctly
33     if img is None:
34         print(f"Error loading image: {image_filepath}")
35         return
36
37     # Get the dimensions of the image
38     height, width = img.shape[:2] # Only need width and height
39
40     # Define the corner coordinates
41     corners = {
42         "x": [0, width - 1, width - 1, 0], # P, Q, R, S
43         "y": [0, 0, height - 1, height - 1], # P, Q, R, S
44         "filepath": image_filepath,
45     }
46
47     # Print the corner coordinates for logging
48     print(f"Image corners for {image_filepath}:")
49     print(f"P: [{corners['x'][0]}, {corners['y'][0]}]") # Top-left
50     corner

```

```

49     print(f"Q: [{corners['x'][1]}, {corners['y'][1]}]") #
        Top-right corner
50     print(f"R: [{corners['x'][2]}, {corners['y'][2]}]") #
        Bottom-right corner
51     print(f"S: [{corners['x'][3]}, {corners['y'][3]}]") #
        Bottom-left corner
52
53     return corners
54
55
56 # Function to display the coordinates of the points clicked on the
    image
57 def click_event(event, x, y, flags, params):
58     global img, click_count, letters, coordinates
59
60     # checking for left mouse clicks
61     if event == cv2.EVENT_LBUTTONDOWN:
62         if click_count < 4: # Allow only 4 clicks
63             letter = letters[click_count]
64
65             # Displaying the coordinates with a letter
66             coordinate_str = f"{letter}: [{x}, {y}]"
67             print(coordinate_str)
68
69             # Store coordinates
70             coordinates["x"].append(x)
71             coordinates["y"].append(y)
72
73             # Draw a dot at the clicked location
74             dot_color = (0, 0, 255) # Red color for the dot
75             dot_radius = 10 # Radius of the dot
76             cv2.circle(img, (x, y), dot_radius, dot_color, -1) #
                Filled circle
77
78             # Displaying the coordinates and letter on the image
                window
79             font = cv2.FONT_HERSHEY_SIMPLEX
80             font_scale = 2 # Increased font size
81             thickness = 5 # Increased thickness
82             offset = 15
83             cv2.putText(
84                 img,
85                 coordinate_str,
86                 (x + offset, y + offset),
87                 font,
88                 font_scale,
89                 (255, 255, 255),

```

```

90         thickness,
91     )
92     cv2.imshow("image", img)
93
94     click_count += 1 # Increment the click count
95
96     if click_count == 4: # After 4 clicks, disable further
97         clicks
98         cv2.setMouseCallback("image", lambda *args: None) #
99         Disable further clicks
100
101 def annotate_pixel_coordinates(image_filepath):
102     global img, click_count, coordinates
103
104     # Reset the click count and coordinates for each new image
105     click_count = 0
106     coordinates = {"x": [], "y": []} # Reset coordinates
107
108     # Read in the image
109     img = cv2.imread(image_filepath, 1)
110
111     # Check if image is loaded correctly
112     if img is None:
113         print(f"Error loading image: {image_filepath}")
114         return
115
116     # Get the dimensions of the image
117     height, width, channels = img.shape
118     print(f"Image size: Width = {width}, Height = {height},
119           Channels = {channels}")
120
121     # Calculate aspect ratio
122     aspect_ratio = width / height
123
124     # Desired width or height for the window
125     desired_width = 800 # Example: 800 pixels
126     desired_height = int(desired_width / aspect_ratio)
127
128     # Create a named window that allows resizing
129     cv2.namedWindow("image", cv2.WINDOW_NORMAL)
130
131     # Resize the window to maintain the aspect ratio
132     cv2.resizeWindow("image", desired_width, desired_height)
133
134     # Move window to the center of the screen (assuming 1920x1080
135     screen resolution)

```

```

133     screen_width, screen_height = (
134         1920,
135         1080,
136     ) # You can adjust this to your screen resolution
137     window_x = int((screen_width - desired_width) / 2)
138     window_y = int((screen_height - desired_height) / 2)
139     cv2.moveWindow("image", window_x, window_y)
140
141     # Display the image in the resized window
142     cv2.imshow("image", img)
143
144     # Setting mouse handler for the image
145     # and calling the click_event() function
146     cv2.setMouseCallback("image", click_event)
147
148     # Wait for a key to be pressed to exit
149     cv2.waitKey(0)
150
151     # Save the annotated image with the suffix "_annotated"
152     annotated_filepath = image_filepath.replace(".jpg",
153         "_annotated.jpg")
154     cv2.imwrite(annotated_filepath, img)
155     print(f"Annotated image saved as: {annotated_filepath}")
156
157     # Properly close the window and release resources
158     cv2.destroyAllWindows()
159
160     return coordinates # Return the collected coordinates
161
162 def write_coordinates_to_json(image_data, json_filepath):
163     """
164     Writes image coordinates to a JSON file. If the file exists,
165     append new entries.
166
167     :param image_data: Dictionary containing image filepaths and
168         coordinates.
169     :param json_filepath: The path to the JSON file.
170     """
171     if os.path.exists(json_filepath):
172         # If the JSON file exists, read its current content
173         with open(json_filepath, "r") as json_file:
174             existing_data = json.load(json_file)
175     else:
176         existing_data = {}
177
178     # Update existing data with new data

```



```

177     existing_data.update(image_data)
178
179     # Write updated data back to JSON file
180     with open(json_filepath, "w") as json_file:
181         json.dump(existing_data, json_file, indent=4)
182     print(f"Coordinates saved to {json_filepath}")
183
184
185 def task_1(log_filename, json_filename):
186     # Set up logging to both terminal and file
187     logger = Logger(log_filename)
188     sys.stdout = logger
189
190     image1_filepath = "HW2_images/img1.jpg"
191     image2_filepath = "HW2_images/img2.jpg"
192     image3_filepath = "HW2_images/img3.jpg"
193     alex_filepath = "HW2_images/alex_honnold.jpg"
194
195     image_data = {} # To store data for all images
196
197     # Annotate images
198     print_with_delimiter(f"Annotating Coordinates for:
199                          {image1_filepath}")
200     img1_coordinates = annotate_pixel_coordinates(image1_filepath)
201     img1_coordinates["filepath"] = image1_filepath
202     image_data["img1"] = img1_coordinates
203     print(f"-----")
204
205     print_with_delimiter(f"Annotating Coordinates for:
206                          {image2_filepath}")
207     img2_coordinates = annotate_pixel_coordinates(image2_filepath)
208     img2_coordinates["filepath"] = image2_filepath
209     image_data["img2"] = img2_coordinates
210     print(f"-----")
211
212     print_with_delimiter(f"Annotating Coordinates for:
213                          {image3_filepath}")
214     img3_coordinates = annotate_pixel_coordinates(image3_filepath)
215     img3_coordinates["filepath"] = image3_filepath
216     image_data["img3"] = img3_coordinates
217     print(f"-----")
218
219     print_with_delimiter(f"Extracting Image Corners for:
220                          {alex_filepath}")
221     alex_corners = print_image_corners(alex_filepath)
222     image_data["alex"] = alex_corners
223     print(f"-----")

```

```

220
221 # Write image data to JSON
222 write_coordinates_to_json(image_data, json_filename)
223
224 # Close the log file and reset stdout
225 logger.close()
226 sys.stdout = logger.terminal
227 print(f"--- Log saved to {log_filename}")
228
229
230 def task_2(log_filename, json_filename):
231     # Set up logging to both terminal and file
232     logger = Logger(log_filename)
233     sys.stdout = logger
234
235     bear1_filepath = "HW2_images/bear1.jpg"
236     bear2_filepath = "HW2_images/bear2.jpg"
237     bear3_filepath = "HW2_images/bear3.jpg"
238     haggis_filepath = "HW2_images/haggis.png"
239
240     image_data = {} # To store data for all images
241
242     # Annotate images
243     print_with_delimiter(f"Annotating Coordinates for:
244         {bear1_filepath}")
245     bear1_coordinates = annotate_pixel_coordinates(bear1_filepath)
246     bear1_coordinates["filepath"] = bear1_filepath
247     image_data["bear1"] = bear1_coordinates
248     print(f"-----")
249
250     print_with_delimiter(f"Annotating Coordinates for:
251         {bear2_filepath}")
252     bear2_coordinates = annotate_pixel_coordinates(bear2_filepath)
253     bear2_coordinates["filepath"] = bear2_filepath
254     image_data["bear2"] = bear2_coordinates
255     print(f"-----")
256
257     print_with_delimiter(f"Annotating Coordinates for:
258         {bear3_filepath}")
259     bear3_coordinates = annotate_pixel_coordinates(bear3_filepath)
260     bear3_coordinates["filepath"] = bear3_filepath
261     image_data["bear3"] = bear3_coordinates
262     print(f"-----")
263
264     print_with_delimiter(f"Extracting Image Corners for:
265         {haggis_filepath}")
266     haggis_corners = print_image_corners(haggis_filepath)

```

```

263     image_data["haggis"] = haggis_corners
264     print(f"-----")
265
266     # Write image data to JSON
267     write_coordinates_to_json(image_data, json_filename)
268
269     # Close the log file and reset stdout
270     logger.close()
271     sys.stdout = logger.terminal
272     print(f"--- Log saved to {log_filename}")
273
274
275 if __name__ == "__main__":
276     task_1_log = "task_1_log.txt"
277     task_2_log = "task_2_log.txt"
278     json_file = "img_data.json"
279
280     # Obtain pixel coordinates for files from Tasks 1 and 2...
281     task_1(task_1_log, json_file)
282     task_2(task_2_log, json_file)

```

Python Code: Script to help complete the extra credit

```

1  import cv2
2  import numpy as np
3  import os
4  import matplotlib.pyplot as plt
5
6
7  def create_grid_image(width, height, grid_size):
8      """
9      Function to create our desired grid image with an 'arrow-like'
10     structure in the middle (to help determine orientation)
11     """
12
13     # Create a blank white image
14     image = np.ones((height, width), dtype=np.uint8) * 255 # 255
15     for a white image
16
17     # Draw grid lines
18     for i in range(0, width, grid_size):
19         cv2.line(image, (i, 0), (i, height), (0), 1) # Vertical
20         lines
21
22     for j in range(0, height, grid_size):
23         cv2.line(image, (0, j), (width, j), (0), 1) # Horizontal
24         lines

```

```

20
21 # Calculate the center of the grid
22 center_x = (width // 2) // grid_size # Center in grid units
23 center_y = (height // 2) // grid_size # Center in grid units
24
25 # Define the arrow tip and body relative to the center
26 arrow_tip = (center_x, center_y) # Center grid position
27 arrow_body = [
28     (center_x - 1, center_y),
29     (center_x - 2, center_y),
30     (center_x, center_y - 1),
31     (center_x, center_y + 1),
32 ]
33
34 # Draw the arrow tip
35 cv2.rectangle(
36     image,
37     (arrow_tip[0] * grid_size, arrow_tip[1] * grid_size),
38     ((arrow_tip[0] + 1) * grid_size, (arrow_tip[1] + 1) *
39         grid_size),
40     (0),
41     -1,
42 )
43
44 # Draw the arrow body
45 for bx, by in arrow_body:
46     cv2.rectangle(
47         image,
48         (bx * grid_size, by * grid_size),
49         ((bx + 1) * grid_size, (by + 1) * grid_size),
50         (0),
51         -1,
52     )
53
54 return image
55
56 def apply_homography(image, H):
57     """
58     Function to apply a homography matrix to an input image.
59     NOTE: For this part, we are allowed to use
60         cv2.warpPerspective(), unlike the prior parts
61     """
62     height, width = image.shape[:2]
63     return cv2.warpPerspective(image, H, (width, height))
64

```

```

65 def save_images(images, transformation_type, angles):
66     """
67     Function to save our transformed image plots
68     """
69     if not os.path.exists("HW2_images"):
70         os.makedirs("HW2_images")
71
72     for i, img in enumerate(images):
73         cv2.imwrite(
74             f"HW2_images/{transformation_type}_transformation_angle_{angles[i]}.png",
75             img,
76         )
77
78
79 def plot_images(images, transformation_type, angles):
80     """
81     Function to create side-by-side figures that show
82     transformations for all available angles
83     """
84     # Create a figure for the transformation type
85     fig, axes = plt.subplots(1, len(images), figsize=(15, 5))
86     fig.suptitle(f"{transformation_type.capitalize()}
87                 Transformations", fontsize=16)
88
89     for i, (img, angle) in enumerate(zip(images, angles)):
90         axes[i].imshow(img, cmap="gray")
91         axes[i].set_title(f"Angle: {angle:.0f} ")
92         axes[i].axis("off")
93
94     # Save the figure
95     plt.savefig(f"HW2_images/{transformation_type}_transformations.png")
96     plt.show()
97
98 def apply_homography_to_line(H, line):
99     """
100     Applies a homography transformation to a line in homogeneous
101     coordinates.
102
103     Inputs:
104     - H: 3x3 numpy array representing the homography matrix.
105     - line: 3x1 numpy array (column vector) representing the line
106         in homogeneous coordinates ( $ax + by + c = 0$ ).
107
108     Returns "transformed_line": 3x1 numpy array (column vector)
109         representing the transformed line.
110     """

```

```

107     # Compute the inverse transpose of the homography matrix
108     H_inv_transpose = np.linalg.inv(H).T
109
110     # Apply the transformation
111     transformed_line = H_inv_transpose @ line
112
113     return transformed_line
114
115
116 def normalize_line(line):
117     """
118     Normalizes a line in homogeneous coordinates by dividing by the
119         third coordinate.
120
121     Inputs:
122     - line: 3x1 numpy array (column vector) representing the line
123         in homogeneous coordinates.
124
125     Returns "normalized_line": 3x1 numpy array (column vector)
126         representing the normalized line.
127     """
128     return line / line[2]
129
130 if __name__ == "__main__":
131     # Define the image dimensions and grid size
132     width, height = 500, 500
133     grid_size = 50 # Size of the grid cells
134
135     # Create the grid image
136     input_image = create_grid_image(width, height, grid_size)
137     cv2.imwrite(f"HW2_images/original_grid.png", input_image)
138
139     # Define the list of angles in degrees
140     angles = [15, 45, -45]
141
142     # Normalization and de-normalization homographies
143     H_norm = np.array([[2 / width, 0, -1], [0, 2 / height, -1], [0,
144         0, 1]])
145
146     H_de_norm = np.array(
147         [[width / 2, 0, width / 2], [0, height / 2, height / 2],
148         [0, 0, 1]]
149     )
150
151     # Lists to store output images for each transformation type

```

```

149 rotated_images = []
150 horz_tilt_images = []
151 vert_tilt_images = []
152
153 for angle in angles:
154     alpha = np.radians(angle) # Convert degrees to radians
155
156     # Rotation homography matrix
157     H_rot = np.array(
158         [
159             [np.cos(alpha), -np.sin(alpha), 0],
160             [np.sin(alpha), np.cos(alpha), 0],
161             [0, 0, 1],
162         ]
163     )
164
165     # Horizontal tilting homography matrix
166     H_horz_tilt = np.array(
167         [
168             [np.cos(alpha), 0, -np.sin(alpha)],
169             [0, 1, 0],
170             [np.sin(alpha), 0, np.cos(alpha)],
171         ]
172     )
173
174     # Vertical tilting homography matrix
175     H_vert_tilt = np.array(
176         [
177             [1, 0, 0],
178             [0, np.cos(alpha), np.sin(alpha)],
179             [0, -np.sin(alpha), np.cos(alpha)],
180         ]
181     )
182
183     # Composite homographies
184     H_composite_rot = H_de_norm @ H_rot @ H_norm
185     H_composite_horz_tilt = H_de_norm @ H_horz_tilt @ H_norm
186     H_composite_vert_tilt = H_de_norm @ H_vert_tilt @ H_norm
187
188     # Apply the transformations
189     rotated_image = apply_homography(input_image,
190                                     H_composite_rot)
191     horz_tilt_image = apply_homography(input_image,
192                                       H_composite_horz_tilt)
193     vert_tilt_image = apply_homography(input_image,
194                                       H_composite_vert_tilt)

```

```

193     # Append the results to the corresponding lists
194     rotated_images.append(rotated_image)
195     horz_tilt_images.append(horz_tilt_image)
196     vert_tilt_images.append(vert_tilt_image)
197
198 # Save images
199 save_images(rotated_images, "rotated", angles)
200 save_images(horz_tilt_images, "horizontal_tilt", angles)
201 save_images(vert_tilt_images, "vertical_tilt", angles)
202
203 # Plot and save figures
204 plot_images(rotated_images, "rotated", angles)
205 plot_images(horz_tilt_images, "horizontal_tilt", angles)
206 plot_images(vert_tilt_images, "vertical_tilt", angles)
207
208 print("Images saved and figures created successfully.")
209
210 """
211 Extra Credit Problem 3
212 """
213 line_x_parallel = np.array([0, 1, -1]).reshape(3, 1)
214 line_y_parallel = np.array([1, 0, -1]).reshape(3, 1)
215
216 alpha = 15
217 alpha_rad = np.radians(alpha)
218
219 # Horizontal tilting homography matrix
220 H_horz_tilt = np.array(
221     [
222         [np.cos(alpha_rad), 0, -np.sin(alpha_rad)],
223         [0, 1, 0],
224         [np.sin(alpha_rad), 0, np.cos(alpha_rad)],
225     ]
226 )
227
228 # Vertical tilting homography matrix
229 H_vert_tilt = np.array(
230     [
231         [1, 0, 0],
232         [0, np.cos(alpha_rad), np.sin(alpha_rad)],
233         [0, -np.sin(alpha_rad), np.cos(alpha_rad)],
234     ]
235 )
236
237 # Apply homography to both lines
238
239 # Horizontal Tilt

```



```

240 print(f"\n=====")
241 print(f"Applying Horizontal Tilt: {alpha} degrees")
242 print(f"=====")
243 horz_tilt_x_parallel = apply_homography_to_line(H_horz_tilt,
        line_x_parallel)
244 horz_tilt_y_parallel = apply_homography_to_line(H_horz_tilt,
        line_y_parallel)
245
246 print(f"Horizontally tilted x_parallel (UNNORMALIZED):
        {horz_tilt_x_parallel.T}")
247 print(f"Horizontally tilted y_parallel (UNNORMALIZED):
        {horz_tilt_y_parallel.T}")
248
249 # Normalize the results
250 horz_tilt_x_parallel = normalize_line(horz_tilt_x_parallel)
251 horz_tilt_y_parallel = normalize_line(horz_tilt_y_parallel)
252
253 print(f"Horizontally tilted x_parallel:
        {horz_tilt_x_parallel.T}")
254 print(f"Horizontally tilted y_parallel:
        {horz_tilt_y_parallel.T}")
255
256 # Vertical Tilt
257 print(f"\n=====")
258 print(f"Applying Vertical Tilt: {alpha} degrees")
259 print(f"=====")
260 vert_tilt_x_parallel = apply_homography_to_line(H_vert_tilt,
        line_x_parallel)
261 vert_tilt_y_parallel = apply_homography_to_line(H_vert_tilt,
        line_y_parallel)
262
263 print(f"Vertically tilted x_parallel (UNNORMALIZED):
        {vert_tilt_x_parallel.T}")
264 print(f"Vertically tilted y_parallel (UNNORMALIZED):
        {vert_tilt_y_parallel.T}")
265
266 # Normalize the results
267 vert_tilt_x_parallel = normalize_line(vert_tilt_x_parallel)
268 vert_tilt_y_parallel = normalize_line(vert_tilt_y_parallel)
269
270 print(f"Vertically tilted x_parallel: {vert_tilt_x_parallel.T}")
271 print(f"Vertically tilted y_parallel: {vert_tilt_y_parallel.T}")

```

Python Code: Helper functions and classes

```

1 import sys

```

```

2 import cv2
3 import math
4 import tkinter as tk
5
6
7 # class to print both to terminal and log file
8 class Logger:
9     def __init__(self, log_file):
10         self.terminal = sys.stdout # Save the original stdout
11         self.log_file = open(log_file, "w")
12
13     def write(self, message):
14         self.terminal.write(message) # Print to terminal
15         self.log_file.write(message) # Write to log file
16
17     def flush(self):
18         self.terminal.flush() # Ensure terminal flushes its buffer
19         self.log_file.flush() # Ensure file flushes its buffer
20
21     def close(self):
22         self.log_file.close() # Close the file
23
24
25 def center_window(window_name, image):
26     """
27     Centers an OpenCV window on the screen
28
29     Inputs:
30         window_name: The name of the OpenCV window._
31         image: The image displayed in the window, needed to get its
32             size.
33     """
34
35     # Initialize tkinter to fetch screen size
36     root = tk.Tk()
37     screen_width = root.winfo_screenwidth()
38     screen_height = root.winfo_screenheight()
39     root.destroy() # Close the tkinter window
40
41     # Get image dimensions
42     image_height, image_width = image.shape[:2]
43
44     # Calculate the position to center the window
45     x_pos = math.floor((screen_width - image_width) // 2)
46     y_pos = math.floor((screen_height - image_height) // 2.5)
47
48     # Move the OpenCV window to the calculated position

```

```

48     cv2.moveWindow(window_name, x_pos, y_pos)
49
50
51 def print_with_delimiter(text, delimiter="*"):
52     """
53     Prints the given text surrounded by a specified delimiter,
54     with lines of the delimiter above and below the text.
55
56     Inputs:
57         text: The string to be surrounded and printed.
58         delimiter: The character(s) to surround and print the lines.
59     """
60     # Create the line of delimiters
61     delimiter_line = delimiter * (len(text) + 4) # +4 for the
        delimiters on either side
62
63     # Print the delimiter line above
64     print(f"\n{delimiter_line}")
65
66     # Print the text surrounded by delimiters
67     print(f"{delimiter} {text} {delimiter}")
68
69     # Print the delimiter line below
70     print(delimiter_line)
71
72
73 import numpy as np
74
75
76 def print_threeXthree_array(array, decimals=3):
77     """
78     Nicely prints a 3x3 NumPy array with formatted alignment.
79
80     Inputs:
81         array (np.ndarray): A 3x3 NumPy array to print.
82         decimals (int): Number of decimal places to format the
            numbers.
83     """
84     # Check if the input array is 3x3
85     if array.shape != (3, 3):
86         raise ValueError("Input array must be a 3x3 NumPy array.")
87
88     # Use NumPy's set_printoptions to control formatting
89     with np.printoptions(precision=decimals, suppress=True):
90         for row in array:
91             print(" ".join(f"{val:8.3f}" for val in row))

```

JSON File: HW2 Image Data

```
1 {
2   "img1": {
3     "x": [
4       424,
5       2547,
6       2399,
7       684
8     ],
9     "y": [
10      856,
11      880,
12      2290,
13      3171
14    ],
15    "filepath": "HW2_images/img1.jpg"
16  },
17  "img2": {
18    "x": [
19      524,
20      1866,
21      1950,
22      464
23    ],
24    "y": [
25      1441,
26      820,
27      2810,
28      2690
29    ],
30    "filepath": "HW2_images/img2.jpg"
31  },
32  "img3": {
33    "x": [
34      1217,
35      2971,
36      1790,
37      248
38    ],
39    "y": [
40      560,
41      2302,
42      3179,
43      1801
44    ],
45    "filepath": "HW2_images/img3.jpg"
```

```

46 },
47 "alex": {
48     "x": [
49         0,
50         782,
51         782,
52         0
53     ],
54     "y": [
55         0,
56         0,
57         664,
58         664
59     ],
60     "filepath": "HW2_images/alex_honnold.jpg"
61 },
62 "bear1": {
63     "x": [
64         600,
65         2030,
66         2030,
67         728
68     ],
69     "y": [
70         1045,
71         1185,
72         2346,
73         2798
74     ],
75     "filepath": "HW2_images/bear1.jpg"
76 },
77 "bear2": {
78     "x": [
79         961,
80         2271,
81         2246,
82         989
83     ],
84     "y": [
85         1417,
86         1321,
87         2730,
88         2454
89     ],
90     "filepath": "HW2_images/bear2.jpg"
91 },
92 "bear3": {

```

```

93         "x": [
94             720,
95             2623,
96             1670,
97             128
98         ],
99         "y": [
100             760,
101             1573,
102             3375,
103             2162
104         ],
105         "filepath": "HW2_images/bear3.jpg"
106     },
107     "haggis": {
108         "x": [
109             0,
110             1398,
111             1398,
112             0
113         ],
114         "y": [
115             0,
116             0,
117             693,
118             693
119         ],
120         "filepath": "HW2_images/haggis.png"
121     }
122 }

```

Log File: Task 1 PQRS Coordinates

```

*****
* Annotating Coordinates for: HW2_images/img1.jpg *
*****
Image size: Width = 3024, Height = 4032, Channels = 3
P: [424, 856]
Q: [2547, 880]
R: [2399, 2290]
S: [684, 3171]
Annotated image saved as: HW2_images/img1_annotated.jpg

```

```
*****
* Annotating Coordinates for: HW2_images/img2.jpg *
*****
Image size: Width = 3024, Height = 4032, Channels = 3
P: [524, 1441]
Q: [1866, 820]
R: [1950, 2810]
S: [464, 2690]
Annotated image saved as: HW2_images/img2_annotated.jpg
```

```
*****
* Annotating Coordinates for: HW2_images/img3.jpg *
*****
Image size: Width = 3024, Height = 4032, Channels = 3
P: [1217, 560]
Q: [2971, 2302]
R: [1790, 3179]
S: [248, 1801]
Annotated image saved as: HW2_images/img3_annotated.jpg
```

```
*****
* Extracting Image Corners for: HW2_images/alex_honnold.jpg *
*****
Image corners for HW2_images/alex_honnold.jpg:
P: [0, 0]
Q: [782, 0]
R: [782, 664]
S: [0, 664]
```

Coordinates saved to img_data.json

Log File: Task 2 PQRS Coordinates

```
*****
* Annotating Coordinates for: HW2_images/bear1.jpg *
*****
Image size: Width = 3024, Height = 4032, Channels = 3
P: [600, 1045]
Q: [2030, 1185]
R: [2030, 2346]
S: [728, 2798]
Annotated image saved as: HW2_images/bear1_annotated.jpg
```

```
*****
* Annotating Coordinates for: HW2_images/bear2.jpg *
*****
Image size: Width = 3024, Height = 4032, Channels = 3
P: [961, 1417]
Q: [2271, 1321]
R: [2246, 2730]
S: [989, 2454]
Annotated image saved as: HW2_images/bear2_annotated.jpg
```

```
*****
* Annotating Coordinates for: HW2_images/bear3.jpg *
*****
Image size: Width = 3024, Height = 4032, Channels = 3
P: [720, 760]
Q: [2623, 1573]
R: [1670, 3375]
S: [128, 2162]
Annotated image saved as: HW2_images/bear3_annotated.jpg
```

```
*****
* Extracting Image Corners for: HW2_images/haggis.png *
*****
Image corners for HW2_images/haggis.png:
P: [0, 0]
Q: [1398, 0]
R: [1398, 693]
S: [0, 693]
```

Coordinates saved to img_data.json

Environment File: Python packages in conda environment

```
name: ece661
channels:
  - conda-forge
  - defaults
dependencies:
  - _libgcc_mutex=0.1=conda-forge
```


- `_openmp_mutex=4.5=2_gnu`
- `asttokens=2.4.1=pyhd8ed1ab_0`
- `bzip2=1.0.8=h5eee18b_6`
- `ca-certificates=2024.8.30=hbcca054_0`
- `comm=0.2.2=pyhd8ed1ab_0`
- `debugpy=1.6.7=py312h6a678d5_0`
- `decorator=5.1.1=pyhd8ed1ab_0`
- `exceptiongroup=1.2.2=pyhd8ed1ab_0`
- `executing=2.0.1=pyhd8ed1ab_0`
- `expat=2.6.2=h6a678d5_0`
- `importlib-metadata=8.4.0=pyha770c72_0`
- `importlib-metadata=8.4.0=hd8ed1ab_0`
- `ipykernel=6.29.5=pyh3099207_0`
- `ipython=8.27.0=pyh707e725_0`
- `jedi=0.19.1=pyhd8ed1ab_0`
- `jupyter_client=8.6.2=pyhd8ed1ab_0`
- `jupyter_core=5.7.2=py312h06a4308_0`
- `ld_impl_linux-64=2.38=h1181459_1`
- `libffi=3.4.4=h6a678d5_1`
- `libgcc=14.1.0=h77fa898_1`
- `libgcc-ng=14.1.0=h69a702a_1`
- `libgomp=14.1.0=h77fa898_1`
- `libsodium=1.0.18=h36c2ea0_1`
- `libstdcxx-ng=11.2.0=h1234567_1`
- `libuuid=1.41.5=h5eee18b_0`
- `matplotlib-inline=0.1.7=pyhd8ed1ab_0`
- `ncurses=6.4=h6a678d5_0`
- `nest-asyncio=1.6.0=pyhd8ed1ab_0`
- `openssl=3.3.1=hb9d3cd8_3`
- `packaging=24.1=pyhd8ed1ab_0`
- `parso=0.8.4=pyhd8ed1ab_0`
- `pexpect=4.9.0=pyhd8ed1ab_0`
- `pickleshare=0.7.5=py_1003`
- `pip=24.2=py312h06a4308_0`
- `platformdirs=4.2.2=pyhd8ed1ab_0`
- `prompt-toolkit=3.0.47=pyha770c72_0`
- `psutil=5.9.0=py312h5eee18b_0`
- `ptyprocess=0.7.0=pyhd3deb0d_0`
- `pure_eval=0.2.3=pyhd8ed1ab_0`
- `pygments=2.18.0=pyhd8ed1ab_0`
- `python=3.12.4=h5148396_1`
- `python-dateutil=2.9.0=pyhd8ed1ab_0`
- `pyzmq=25.1.2=py312h6a678d5_0`
- `readline=8.2=h5eee18b_0`
- `setuptools=72.1.0=py312h06a4308_0`

```

- six=1.16.0=pyh6c4a22f_0
- sqlite=3.45.3=h5eee18b_0
- stack_data=0.6.2=pyhd8ed1ab_0
- tk=8.6.14=h39e8969_0
- tornado=6.4.1=py312h5eee18b_0
- traitlets=5.14.3=pyhd8ed1ab_0
- typing_extensions=4.12.2=pyha770c72_0
- tzdata=2024a=h04d1e81_0
- wcwidth=0.2.13=pyhd8ed1ab_0
- wheel=0.43.0=py312h06a4308_0
- xz=5.4.6=h5eee18b_1
- zeromq=4.3.5=h6a678d5_0
- zipp=3.20.1=pyhd8ed1ab_0
- zlib=1.2.13=h5eee18b_1
- pip:
  - black==24.8.0
  - click==8.1.7
  - contourpy==1.3.0
  - cyclers==0.12.1
  - fonttools==4.53.1
  - imageio==2.35.1
  - kiwisolver==1.4.5
  - lazy-loader==0.4
  - matplotlib==3.9.2
  - mypy_extensions==1.0.0
  - networkx==3.3
  - numpy==2.1.0
  - opencv-python==4.10.0.84
  - pathspec==0.12.1
  - pillow==10.4.0
  - pyparsing==3.1.4
  - scikit-image==0.24.0
  - scipy==1.14.1
  - tifffile==2024.8.28
prefix: /home/gooseneck/anaconda3/envs/ece661

```

References

- [1] Fiction Taxonomy Fandom. *Wild Haggis*. https://fiction-taxonomy.fandom.com/wiki/Wild_haggis. Image used from webpage, Accessed: 2024-09-04.
- [2] GeeksforGeeks. *Displaying the Coordinates of the Points Clicked on the Image using Python-OpenCV*. <https://www.geeksforgeeks.org/displaying-the-coordinates-of-the-points-clicked-on-the-image-using-python-opencv/>. Accessed: 2024-09-04.
- [3] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. 4th ed. Harlow, Essex, England: Pearson Education Limited, 2018. ISBN: 978-1-292-22304-9.
- [4] NumPy Developers. *NumPy: Linear algebra (numpy.linalg.lstsq)*. Accessed: 2024-09-04. 2024. URL: <https://numpy.org/doc/stable/reference/generated/numpy.linalg.lstsq.html>.
- [5] NumPy Developers. *NumPy: Linear algebra (numpy.linalg.solve)*. Accessed: 2024-09-04. 2024. URL: <https://numpy.org/doc/stable/reference/generated/numpy.linalg.solve.html>.
- [6] NumPy Developers. *numpy.rint*. Online; accessed 2024-09-04. 2023. URL: <https://numpy.org/doc/stable/reference/generated/numpy.rint.html>.