ECE 364 Project: Homography

Phase I

Due: November 23, 2015

Image Projective Transformation

Completing this project phase will satisfy course objectives CO2, CO3 and CO6

Instructions

- Work in your Lab11 directory.
- There are no files to copy for this lab.
- This is the first of two phases for the course project for ECE 364. Each phase is worth 12% of your overall course grade.
- This is an **individual** project. All submissions will be checked for plagiarism using an online service.
- Remember to add and commit all files to SVN. Also remember to use your header file to start all scripts. **We will grade the version of the file that is in SVN!**
- Name and spell your scripts exactly as instructed. When you are required to generate output, make sure it matches the specifications exactly, since your scripts will be graded by a computer.
- The use of PyCharm is required, specially to generate the unit test report. **Your submission must include the code file(s) requested and the unit test report.**
- Make sure you are using Python 3.4 for your project. To verify/modify the Python version, go to:

  File Menu → Settings → Project Interpreter

  And make sure that Python 3.4 (`/usr/local/bin/python3.4`) is selected.

Introduction

A projective transformation, or a homography, is a process that applies translation, rotation and scaling to a plane to transform it into a different plane. Figure 1 shows different transformations applied to a plane[1]. This process works by using an invertible 3×3 matrix, referred to as the homography matrix, and applying it to the plane coordinates to transform it from one system to another.

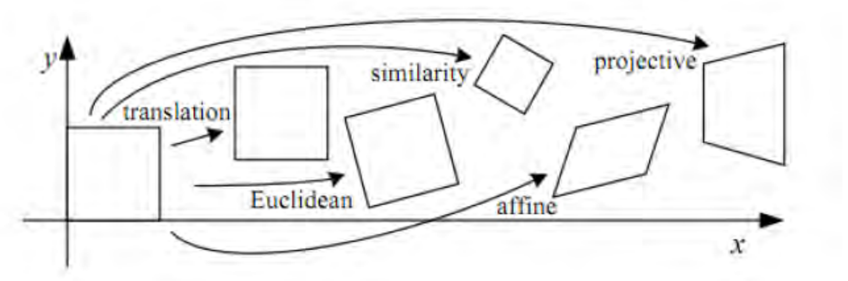


Figure 1: Examples of different transformations on an image.

In this project, you will apply homographies on images and combine them with other container images to produce desired visual effects. This project phase is divided into three stages that build on each other: First, you will apply a given homography to an image, second, you will derive different homographies based on point correspondences from images, and third you will apply custom effects based on correspondences' manipulation.

Stage I Information

Python Modules

This project utilizes three popular Python modules: `numpy`, `scipy` and `pillow`[2]. These modules have excellent documentation and community support, and while this project will use basic functionality from these modules, you are strongly encouraged to investigate them extensively, as you almost surely will need to work with them in any Python-related scientific code-base. You will need to update your account with the latest version of these libraries, which can be done from within PyCharm. (Please note that updating these libraries can take several minutes, so please do that as soon as possible.) Your work in this project will primarily be with the object `ndarray` from the module `numpy`.

Accessing Image Data

In its simplest form, an image is a 2D Array of pixels. (Review the module `scipy.misc` for information on loading an image into a 2D array.) In gray-scale images, a pixel is represented by a single byte, i.e. it takes an integer value between 0 and 255, while in color images, each pixel is represented by three bytes, one for Red, Green and Blue respectively, and each one can take a value between 0 and 255 as well. The colors in the image are also referred to as "channels" or "Bands". Without loss of generality, we will be discussing how to work with gray-scale images.

Since the mathematical foundation of homography is based on plane-to-plane transformation, it is useful to think of an image as a surface on a xy-plane, where at each specific (x, y) coordinates you get $f(x, y)$ which corresponds to the pixel value at the (x, y) indices. As shown in figure 2, the convention is for the origin to

[1]Image obtained from StackOverflow: http://stackoverflow.com/questions/22032618/transformation-concept-in-opencv

[2]There is no specific functionality that we will need the module 'pillow' for, but it is needed for image reading and writing performed by scipy.

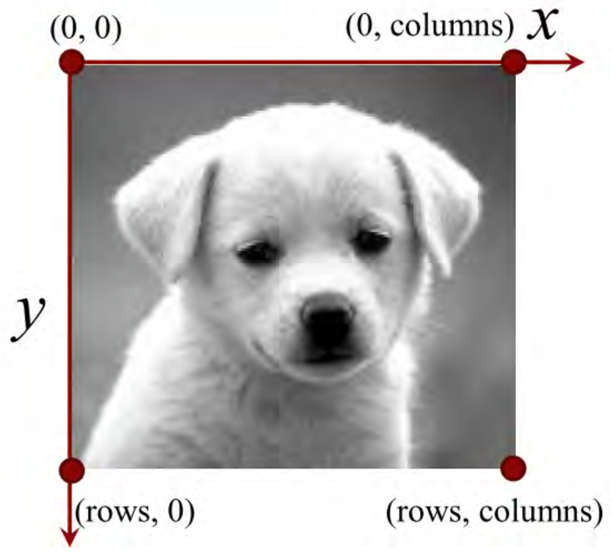


Figure 2: An image can be accessed either by row and column indices, or by x and y coordinates.

be in the upper-left corner of the image, which helps align the x and y coordinates with the row and column indices of the image such that x is the column value, and y is the row value.

Note that when accessing an image by row and column indices, it is indexed as a matrix, i.e. it is accessed as `img[row, column]`, but when it is accessed using the Cartesian system, it is accessed as $f(x, y)$. This distinction is crucial to keep in mind; because based on what accessor method you are using, the order might be reversed. (Please refer to the documentation of the library function you are using to identify the convention of that function.)

While image values are defined at integer indices, viewing the image as a surface allows us to access the image at non-integer coordinates, like at $x = 3.5, y = 2.75$. We can obtain the image value at such locations by utilizing 2D interpolation to approximately calculate that value from its 4 neighbors. There are different mathematical approaches to performing 2D interpolation, the simplest of which is bilinear interpolation[3]. For this project, you may implement your own bilinear interpolation function, or use one of the functions in the Python modules you are working with, such as:

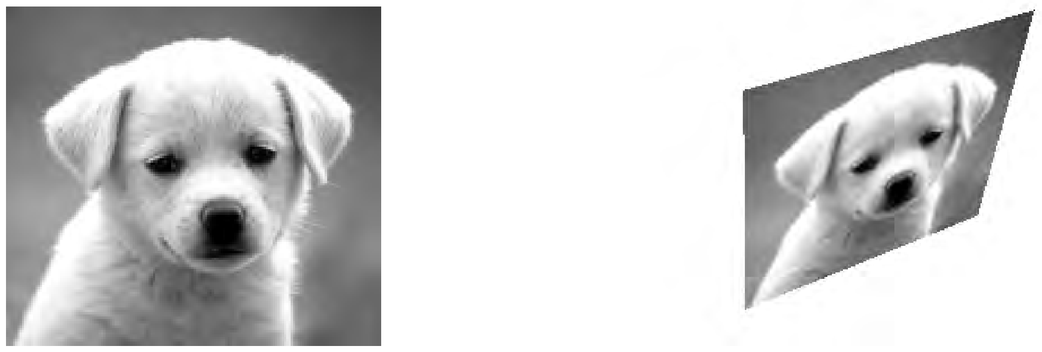
- `scipy.interpolate.RectBivariateSpline()`, which can perform polynomial approximation of any degree. This is the preferred function; because it is much faster than the rest. (If you decide to use this function, you must set both parameters `"kx"` and `"ky"` to 1.)
- `scipy.interpolate.interp2d()`, which is fairly straightforward to use, but is slower than the previous one.

Homography Matrix and Homogeneous Coordinates

Given a homography matrix, H, we can transform an image from one plane to another, as shown in Figure 3. Note that this transformation process is invertible, i.e. if we use the homography matrix H to transform image A into image B, we can use the inverse homography matrix H^{-1} to transform B back into A[4]. In order to apply a homography to a given image, we are going to utilize homogeneous coordinate, which simply

[3]For a simple interoduction to bilinear interpolation, please refer to the Wikipedia page on the topic at https://en.wikipedia.org/wiki/Bilinear_interpolation

[4]Note that this is true only in theory, because in practice we will incur numerical errors from matrix multiplication.



(a) Source Image (b) Target Image

Figure 3: Applying a homography to an image to produce a perspective effect.

augments a point in a plane to be a point in space by adding a third coordinate value, and setting it to 1.

By doing that, each point becomes a 3×1 column vector at which we can multiply it with the homography matrix to perform the transformation. For example, if we pick any point (x, y) in the original image, we can transform it to (x', y') in the target image using the homography H as follows:

$$H. \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} . \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x'_1 \\ x'_2 \\ x'_3 \end{bmatrix} = \begin{bmatrix} x'_1/x'_3 \\ x'_2/x'_3 \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

Of course, to transform the whole image we need to perform this operation on every single point in that image. However, there is a caveat that we need to worry about: when iterating over the source image, we will always be reading the point (x, y) from the original image using integer indices. But, the resultant point (x', y') from the multiplication will always contain non-integer values, so where do we assign it in a integer-based matrix (the target image)? Unlike accessing an image using non-integer Cartesian values, there is no proper way to perform non-integer assignment. If we perform rounding to obtain integer indices we will face undesirable consequences, where multiple points from the source will map to a single point in the target ... etc.

To avoid this issue, we use the notion that homography is an invertible process, and we iterate over the target image points to get the source point using the inverse homography matrix:

$$H^{-1}. \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} x_1/x_3 \\ x_2/x_3 \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

By doing so, we can restrict our access in the target image to go over the integer indices (x', y'), and we obtain the point (x, y) in the source image that we have to access to assign it back to the target. If the source point (x, y) contains non-integer values, we can simply use 2D interpolation as explained in the previous section.

Stage I Requirements

Given the information provided so far, your first task is to implement the application of a homography. Create a Python file named `Homography.py` that should consists only of one or more "**classes**", and, optionally, a conditional main block, (i.e. `if __name__ == "__main__":`). Do not include loose Python statements, but you can, however, write within any class, any number of additional member functions and member variables that you might need.

Homography Class

Implement the `Homography` class that holds the homography matrix details and the logic that is associated with it. The member definitions below represent the public interface that your class should conform to. You **will** need to implement additional functions, and include other member variables to simplify your code.

Member Functions:

- `__init__(self, **kwargs)`:

  Initialize the instance of the `Homography` class. The only accepted input parameter at this stage is:

  - `homographyMatrix` which is a list of three elements, where each element is in turn a list of three float values.

  Raise a `ValueError` with an appropriate message if the parameter passed has a different name, if the values are not of type float, or if the matrix is not of size 3×3.

- `forwardProject(self, point)`:

  Apply the homography to the given (x, y) point, and return the point (x', y'). This method takes in and returns a two-element tuple.

- `inverseProject(self, pointPrime)`:

  Apply the inverse homography to the given (x', y') point, and return the point (x, y). This method takes in and returns a two element tuple.

Transformation Class

Implement the `Transformation` class that manages the mechanics of applying a homography. It holds a source image, and transforms it based on the homography provided to a given target image.

The member definitions below represent the public interface that your class should conform to. You **will** need to implement additional functions, and include other member variables to simplify your code.

Member Functions:

- `__init__(self, sourceImage, homography)`:

  Initialize an instance of the `Transformation` class using a specific `sourceImage`, which is an instance of `ndarray` and a homography instance.

  Raise a `TypeError` with an appropriate message if either of the parameters is not an instance of the correct type.

- `setupTransformation(self, targetPoints)`:

  Takes in `targetPoints`, a list of 4 elements, each of which is a point tuple. These points can be used to identify the range of iteration in the target image[5].

[5]For simplicity, we opted to use these points to identify the bounding box in the target image. In a real application, you only need the source image and the homography matrix. You can forward project the bounds of the source image, and this will give you the bounding box in the target image.

- `transformImage(self, containerImage)`:

  Takes in `containerImage`, an instance of `ndarray`, which is the homography target image. This method uses the homography provided to transform the source image onto the container image, then returns the result. (Note that we are referring to the target image as a container to indicate that it is not empty.) The steps that need to be performed by this method are:

  1. Identify the target bounding box.
  2. For every point within the box, perform an inverse projection of the coordinates.
  3. If the result of the inverse projection falls within the bounds of the source image, read that value (potentially using 2D interpolation.)
  4. If the result falls outside the source image, ignore that value.

  Raise a `TypeError` with an appropriate message if the parameter is not of the correct type, and raise `ValueError` if it does not have exactly two dimensions.

Stage II Information

Homography Computation

Computing a homography involves solving a system of equations using linear algebra. In order to construct the system of equations, we need to identify four point correspondences, as shown in Figure 4, where each correspondence is a pair of points.

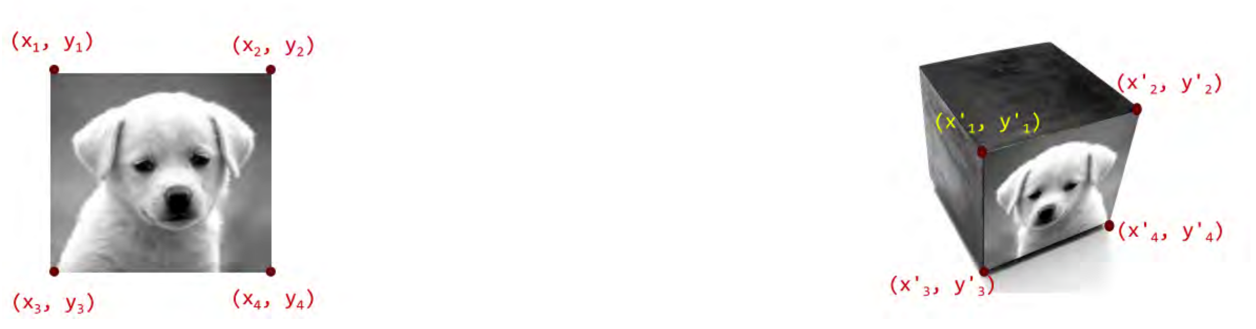


Figure 4: Point correspondences for homography computation.

Intuitively, this process aims to find the matrix that transforms the point (x_1, y_1) from the source image to the point (x'_1, y'_1) in the target image, the point (x_2, y_2) to the point (x'_2, y'_2) and so on. The mathematical derivation of this process is beyond the scope of this project, but the implementation is straightforward.

Each point pair will be used to provide us with the following two matrices:

$$A_n = \begin{bmatrix} x_n & y_n & 1 & 0 & 0 & 0 & -x'_n x_n & -x'_n y_n \\ 0 & 0 & 0 & x_n & y_n & 1 & -y'_n x_n & -y'_n y_n \end{bmatrix}, b_n = \begin{bmatrix} x'_n \\ y'_n \end{bmatrix}$$

For example, the correspondence pair (x_1, y_1) and (x'_1, y'_1) will give the matrices:

$$A_1 = \begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x'_1 x_1 & -x'_1 y_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y'_1 x_1 & -y'_1 y_1 \end{bmatrix}, b_1 = \begin{bmatrix} x'_1 \\ y'_1 \end{bmatrix}$$

With four pairs available, we can stack all of the A_n matrices, and all of the b_n matrices to obtain the system of equations $Ah = b$, where A is an 8×8 matrix, h and b are both 8×1 column vectors:

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x'_1x_1 & -x'_1y_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y'_1x_1 & -y'_1y_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n & y_n & 1 & 0 & 0 & 0 & -x'_nx_n & -x'_ny_n \\ 0 & 0 & 0 & x_n & y_n & 1 & -y'_nx_n & -y'_ny_n \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} = \begin{bmatrix} x'_1 \\ y'_1 \\ \vdots \\ x'_n \\ y'_n \end{bmatrix}$$

which can be solved to obtain the column vector h using the function:

```
h = numpy.linalg.solve(A, b)
```

Finally, we rearrange the column vector into the homography matrix H:

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix}$$

Stage II Requirements

Modify the classes implemented in the previous stage as follows:

Homography Class

We now need to extend the capability of this class to either use a given homography, or compute one from correspondences.

Member Functions:

- `__init__(self, **kwargs)`:

  Modify the initializer to allow for the following keywords in addition to what was already defined:

  – `sourcePoints` and `targetPoints`, each of which is a list of 4 elements, each element is a point tuple. If the class was instantiated with these keywords, invoke the method `computeHomography()` to calculate the homography immediately.

  Raise a `ValueError` with an appropriate message if the parameters passed have different names, if one is missing, or if either contains less than 4 elements.

- `computeHomography(self, sourcePoints, targetPoints)`:

  Takes in two lists, each of which is a list of 4 elements, each element is a point tuple, then computes and returns the homography from the given correspondences.

Transformation Class

As with `Homography`, this class needs to be extended to allow for homography computation.

Member Functions:

- `__init__(self, sourceImage, homography=None)`:

  Modify the initializer to have the `homography` parameter defaulted to `None`.

  Raise a `TypeError` with an appropriate message if the parameter `homography` is not an instance of the class `Homography`, *only* if it is not `None`.

- `setupTransformation(self, targetPoints)`:

  If this instance was not provided a homography in the initializer, use the `targetPoints` passed to instantiate a new `Homography` instance and compute a new homography.

Note that the method `transformImage()` will not need to be modified; because it operates with homography matrix, regardless whether it was obtained to the `Transformation` instance, or calculated from the target points.

Stage III Information

Transformation Effects

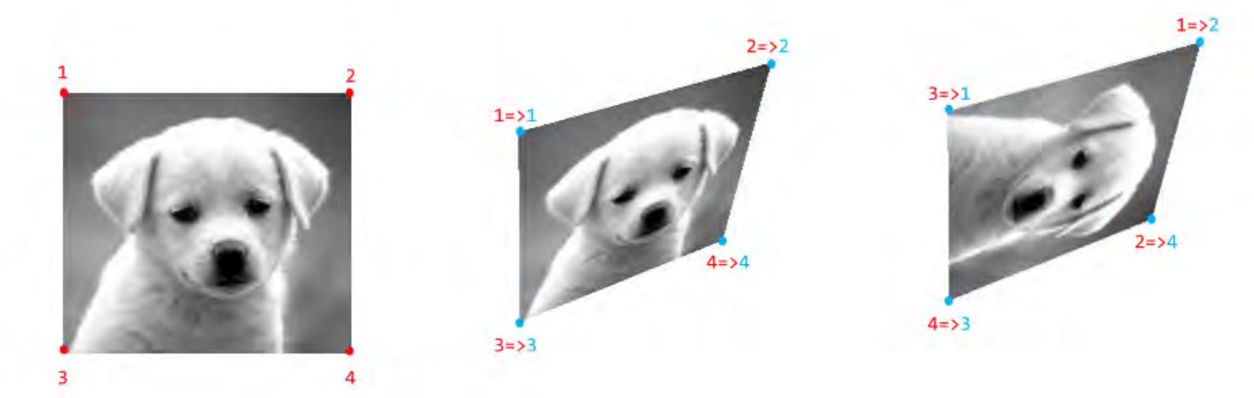
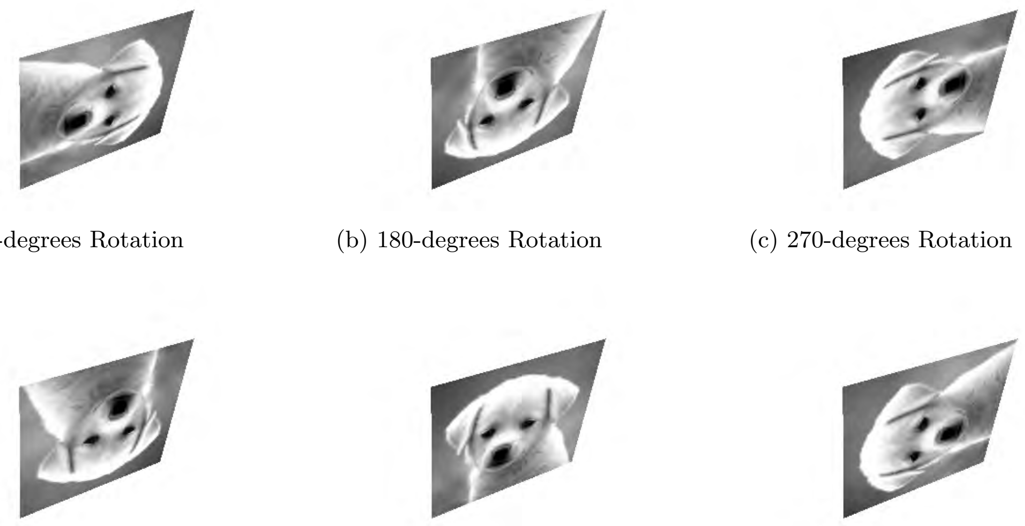


Figure 5: Point correspondences for homography computation.

One outcome of the homography computation is that we can achieve some interesting effects simply by re-ordering the correspondence pairs. In Figure 5, the left image shown is the source image, and the one in the middle shows a standard homography. The image on the right, however, shows a transformed version of the original image that has been rotated 90 degrees clockwise, which is achieved by changing the pairing of the correspondences before we compute the homography. By exploiting different pairing arrangements, we can obtain all of the effects shown in Figure 6. In this part of the project, you will extend your code further more to allow for these effects.



(a) 90-degrees Rotation (b) 180-degrees Rotation (c) 270-degrees Rotation

(d) Horizontal Flip (e) Vertical Flip (f) Transpose

Figure 6: Different effects from homography computation.

Stage III Requirements

Modify the classes implemented in the previous stage as follows:

Effect Enum

Create an `Effect` enum and use it to manage desired effects. This Enum should have the following options:

```
rotate90
rotate180
rotate270
flipHorizontally
flipVertically
transponse
```

Homography Class

Member Functions:

- `__init__(self, **kwargs)`:

  Modify the initializer such that if it is invoked with the keywords `sourcePoints` and `targetPoints`, allow for an "optional" keyword named `effect`. If this keyword is provided, the value must be an instance of `Effect` enumeration.

  Raise a `TypeError` with an appropriate message if `effect` is provided, but it is not an instance of `Effect` enum.

- `computeHomography(self, sourcePoints, targetPoints, effect=None)`:

  Modify this method by including an `effect` parameter that is defaulted to `None`. If it is provided, apply the pair re-ordering *before* you construct the homography matrix to produce the required effect.

Transformation Class

Member Functions:

- `setupTransformation(self, targetPoints, effect=None)`:

  Modify this method by including an `effect` parameter that is defaulted to `None`. You should use this parameter if you are instantiating a new `Homography` instance and computing a new homography.

Homography on Color Images

As described earlier, a color image uses three values per pixel to represent three channels, Red, Green and Blue, or RGB for short. (You can think of a color image as three gray-scale images put together.) Applying homographies to color images would only require you to extend the matrix manipulation to handle three values instead of one. Hence, you will need to only modify one class (if fact, you will need to override only a single method) for your work to extend to color images.

ColorTransformation Class

Inherit and extend the `Transformation` class that you implemented above to include the functionality of applying a homography to a color image.

Member Functions:

- `__init__(self, sourceImage, homography=None)`:

  Initialize an instance of the `ColorTransformation` class using a specific `sourceImage`, which is an instance of `ndarray` and a homography instance. After invoking the base initializer, verify that the image provided is a color image, by checking its dimensions.

  Raise a `ValueError` with an appropriate message if the image provided is not a color image.

- `transformImage(self, containerImage)`:

  Override this method to provide the same functionality as its base, but for color images.

Extra Credit

Now that you have mastered applying homographies to both grayscale and color images, you are required to utilize those skills in implementing two advanced visual effects on images. Figures 7 and 8 show examples of two effects as applied to images. We are going to refer to the one shown in Figure 7 as Effect "V", and to the one shown in Figure 8 as Effect "A"; because the final outcomes resemble the letters "V" and "A". You will only work with color images in this section.

To understand the details of these effects, consider the image in Figure 9 showing the outline of a rectangular image. The points a, b, d, e represent the image corners, and the points m, n represent the midpoints at the boundary. Note that the line connecting m and n is for illustrating the position of these two points. Applying Effect "V" to this rectangular image will transform it to the image shown in Figure 10. This transformation has the following effects[6] on the image:

1. The upper corner points a and d remain where they are, with respect to the original image.
2. The lower corner points b and e move inside by the values v and h_1.
3. The upper midpoint m split into two points, and both move inside by the values v and h_2.

[6] For lack of a better word!

4. The lower midpoint n shifts downwards by the amount v.

Notice how the width of the transformed image remains the same, but the height increase by v. The analysis of Effect "A" follows the same approach, as it affects the other set of points. For both of these effects, the parameters v, h_1 and h_2 are arbitrary, but they are all greater than or equal to 0.

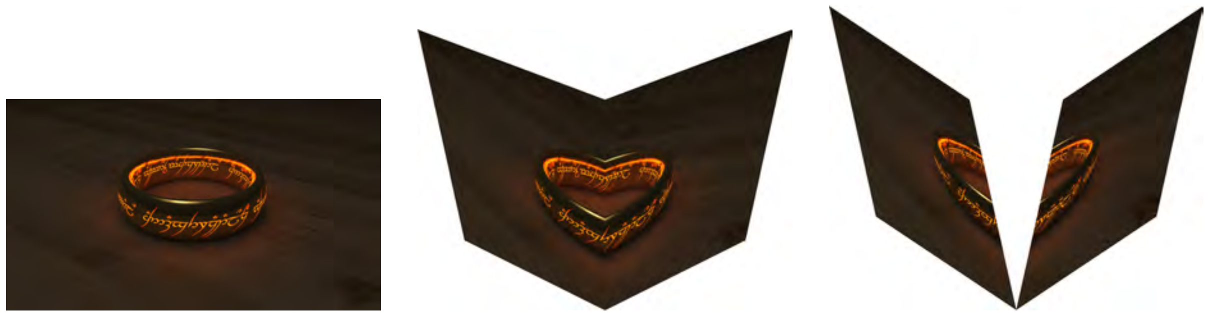


Figure 7: Example of applying Effect "V" to an image.

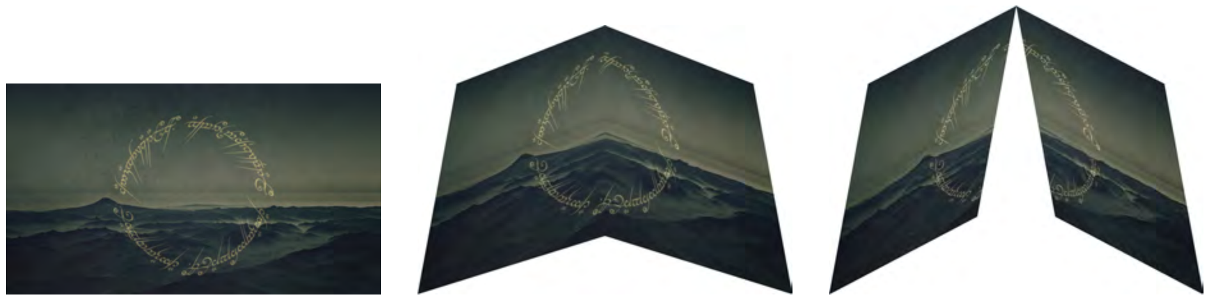


Figure 8: Example of applying Effect "A" to an image.

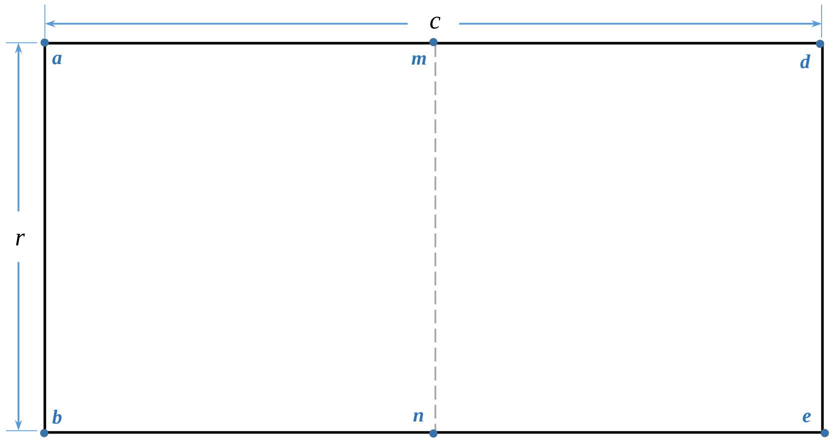


Figure 9: A sample rectangular image.

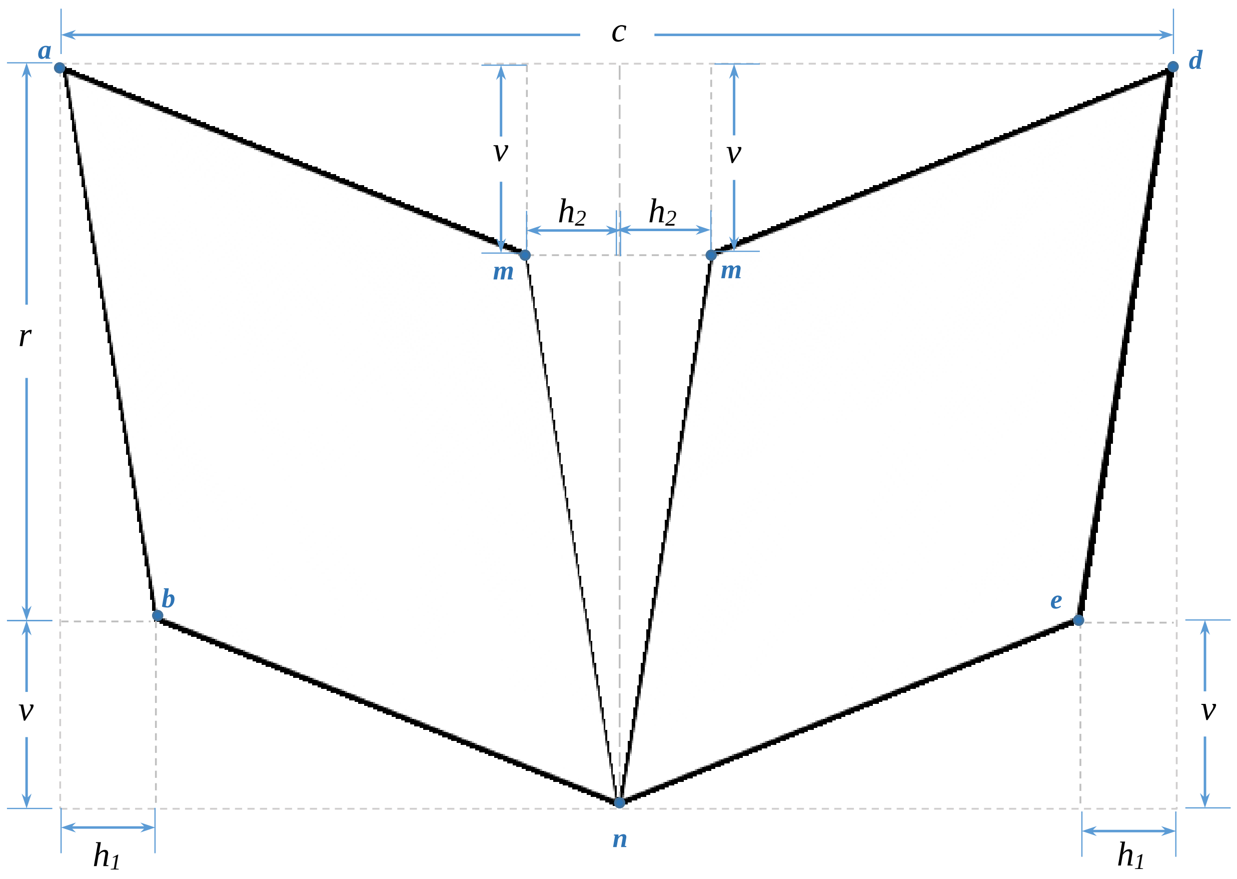


Figure 10: The rectangular image transformed using effect "V".

AdvancedTransformation Class

Create a Python file named `MoreHomography.py` and implement the `AdvancedTransformation` class that contains the operations needed to perform the effects shown above. The member definitions below represent the public interface that your class should conform to. You **will** need to implement additional functions, and include other member variables to simplify your code.

Member Functions:

- `__init__(self, sourceImage, v, h1, h2)`:

  Initialize the instance of the `AdvancedTransformation` class. The parameter `sourceImage` is an instance of `ndarray`, while `v, h1, h2` are all integers greater than or equal to 0.

  Raise a `TypeError` with an appropriate message if the parameter `sourceImage` is not an instance of `ndarray`, and raise a `ValueError` if it does not represent a color image. Additionally, raise a `ValueError` if the number of columns in the image passed is not even.

- `applyEffectV(self)`:

  Apply effect "V" on the image provided to this instance, and return the result.

- `applyEffectA(self)`:

  Apply effect "A" on the image provided to this instance, and return the result.

Verify Your Work!

Remember to run the unit-test file provided, and submit the result report, named as `phase1_results.html`, along with your code.

Appendix: Class Definitions Summary

Here is a recap of the classes used in this project:

Effect Class

An `Enum` class containing the following options:

```
rotate90
rotate180
rotate270
flipHorizontally
flipVertically
transponse
```

Homography Class

A class that manages the homography creation and application.

Member Functions:

- `__init__(self, **kwargs)`

  Acceptable keywords are one of these two groups:

  - `homographyMatrix`.
  - `sourcePoints`, `targetPoints` and optionally `effect`.
- `forwardProject(self, point)`
- `inverseProject(self, pointPrime)`
- `computeHomography(self, sourcePoints, targetPoints, effect=None)`

Transformation Class

A class that manages the source and target images.

Member Functions:

- `__init__(self, sourceImage, homography=None)`
- `setupTransformation(self, targetPoints, effect=None)`
- `transformImage(self, containerImage)`

ColorTransformation Class

A class that manages the source and target images for color images. This class inherits from "Transformation".

Member Functions:

- `__init__(self, sourceImage, homography=None)`
- `transformImage(self, containerImage)`

AdvancedTransformation Class (Optional)

A class that applies advanced visual effects to images.

Member Functions:

- `__init__(self, sourceImage, v, h1, h2)`
- `applyEffectV(self)`
- `applyEffectA(self)`