# 16720: Computer Vision Homework 0 (Scipy Practice)

Instructor: Srinivasa Narasimhan
TAs: Paul Nadan, Cherie Ho, Gautam Gare,
Jack Good, Brady Moon, Ivan Cisneros

**This homework will not be graded**. It is meant to provide some Python/Scipy practice exercises to ensure that coding will not be an obstacle for you during this course. This and future assignments will assume that you are using Python 3+ and the latest Scipy libraries (numpy, matplotlib, etc). The recommended approach is to use the Anaconda Python 3 distribution .

## 1 Color Channel alignment

You have been given the red, green, and blue channels of an image[1] that were taken separately using an old technique that captured each color on a separate piece of glass[2].

These files are named red.npy, green.npy, and blue.npy respectively (in the data folder). Because these images were taken separately, just combining them in a 3-channel matrix may not work. For instance, Figure 1 shows what happens if you simply combine the images without shifting any of the channels.

Your job is to take 3-channel RGB (red-green-blue) image and produce the correct color image (closest to the original scene) as output. Because in the future you may be hired to do this on hundreds of images, you cannot hand-align the image; you must write a function that finds the best possible displacement.

The easiest way to do this is to exhaustively search over a window of possible displacements for the different channels (you can assume the displacement will be between -30 and 30 pixels). Score the alignment from each possible displacement with some heuristic and choose the best alignment using these scores. You can use the following metrics:

1. Sum of Squared Differences (SSD)

   This metric tries to compare the distance between two vectors, hence we can use this to compare image pixel intensity differences. For two vectors **u**, **v** of length

[1] downloaded from http://www.loc.gov/pictures/collection/prok/

[2] Credit to Kris Kitani and Alyosha Efros for this problem.

Figure 1: Combining the red, green, and blue channels without shifting

Figure 2: Aligned image

$N$, this metric is defined as

$$SSD(\mathbf{u}, \mathbf{v}) = \sum_{i=1}^{N} (\mathbf{u}[i] - \mathbf{v}[i])^2$$

(essentially like the Euclidean distance metric).

2. Normalized Cross Correlation (NCC)[3]

   This metric compares normalized unit vectors using a dot product, that is for vectors $\mathbf{u}, \mathbf{v}$,

$$NCC(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u}}{\|\mathbf{u}\|} \cdot \frac{\mathbf{v}}{\|\mathbf{v}\|}$$

Implement an algorithm which finds the best alignment of the three channels to produce a good RGB image; it should look something like Figure 2. Try to avoid using for loops when computing the metrics (SSD or NCC) for a specific offset. Some starter code is given in `script1.py` and `alignChannels.py`. Save your result as `rgb_output.jpg` in the `results` folder.

## 2 Image warping

We will be writing code that performs affine warping on an image. Code has been provided to get you started. In the following sections you will be adding your own code.

### 2.1 Example code

The file `script2.py` contains example code which demonstrates basic image loading and displaying as well as the behavior of an image warping function. Try running `script2`. You should see something like Figure 3. This script calls the function `warp` in `warpA_check.py`, which is simply a wrapper for scipy's own function `scipy.ndimage.affine_transform`.

### 2.2 Affine warp

You will write your own function in `warpA.py`, that should give the same output as the function in `warpA_check.py`. First we'll walk through what you need to know about affine transformations.

An affine transform relates two sets of points:

$$\mathbf{p}^i_{warped} = \underbrace{\begin{pmatrix} a & b \\ c & d \end{pmatrix}}_{\mathbf{L}} \mathbf{p}^i_{source} + \mathbf{t} \tag{1}$$

[3] https://en.wikipedia.org/wiki/Cross-correlation#Normalized_cross-correlation

Figure 3: See script2.py


where $\mathbf{p}^i_{source}$ and $\mathbf{p}^i_{warped}$ denote the 2D coordinates (e.g., $\mathbf{p}^i_s = (x^i_s, y^i_s)^T$) of the $i$-th point in the source space and destination (or warped) space respectively, $\mathbf{L}$ is a $2 \times 2$ matrix representing the linear component, and $\mathbf{t}$ is a $2 \times 1$ vector representing the translational component of the transform.

To more conveniently represent this transformation, we will use homogeneous coordinates, i.e., $\mathbf{p}^i_s$ and $\mathbf{p}^i_w$ will now denote the 2D homogeneous coordinates (e.g., $\mathbf{p}^i_s \equiv (x^i_s, y^i_s, 1)^T$), where "$\equiv$" denotes equivalence up to scale, and the transform becomes:

$$\mathbf{p}^i_d \equiv \underbrace{\left(\begin{array}{cc|c} & \mathbf{L} & \mathbf{t} \\ 0 & 0 & 1 \end{array}\right)}_{\mathbf{A}} \mathbf{p}^i_s \tag{2}$$

- Implement a function that warps image `im` using the affine transform `A`:

  `warp_im = warpA(im, A, output_shape)`

**Inputs**: `im` is a grayscale `double` typed Numpy matrix of dimensions height×width×1 [4], `A` is a $3 \times 3$ non-singular matrix describing the transform ($\mathbf{p}^i_{warped} \equiv \mathbf{A}\mathbf{p}^i_{source}$), and `output_size=[out_height,out_width];` of the warped output image.
**Outputs**: `warp_im` of size `out_size(1)` × `out_size(2)` is the warped output image. The coordinates of the sampled output image points $\mathbf{p}^i_{warped}$ should be the rectangular

[4] Images in Numpy are indexed as `im(row, col, channel)` where `row` corresponds to the $y$ coordinate (height), and `col` to the $x$ coordinate (width).

range $(0, 0)$ to $(width - 1, height - 1)$ of integer values. The points $\mathbf{p}^i_{source}$ must be chosen such that their image, $\mathbf{A}\mathbf{p}^i_{source}$, transforms to this rectangle.

Implementation-wise, this means that we will be looking up the value of each of the destination pixels by sampling the original image at the computed $\mathbf{p}^i_{source}$. (Note that if you do it the other way round, i.e., by transforming each pixel in the source image to the destination, you could get "holes" in the destination because the mapping need not be 1 to 1). In general, the transformed values $\mathbf{p}^i_{source}$ will not lie at integer locations and you will therefore need to choose a sampling scheme; the easiest is nearest-neighbor sampling (something like, `round(`$\mathbf{p}^i_{source}$`)`). You should be able to implement this without using `for` loops (one option is to use `numpy.meshgrid` and Numpy's multidimensional indexing), although it might be easier to implement it using loops first. Save the resulting image in `results/transformed.jpg` .

You should check your implementation to make sure it produces the same output as the `warp` function provided in `warpA_check.py` (for grayscale or RGB images). Obviously the purpose of this exercise is practicing Python/Scipy by implementing your own function without using anything like `scipy.ndimage.affine_transform`.