16720: Computer Vision Homework 0 (Matlab Practice)

Instructor: Martial Hebert
TAs: Varun Ramakrishna and Tomas Simon

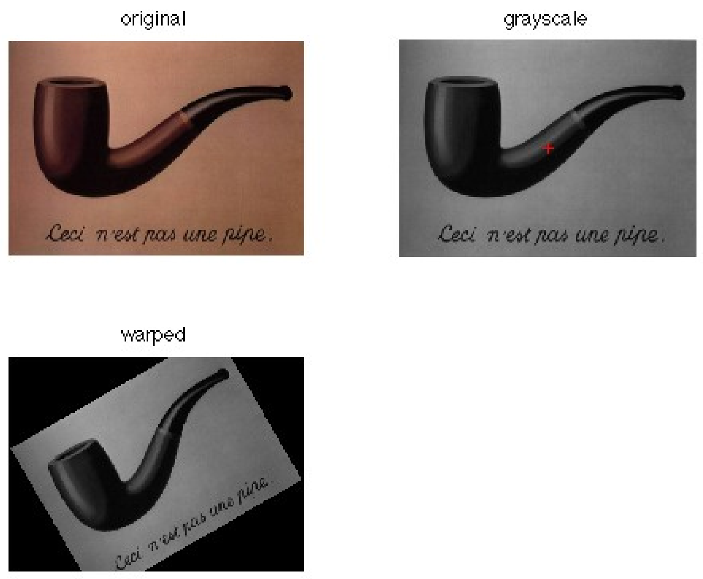
This homework is entirely optional and is only meant to provide some Matlab practice exercises. It will not be graded.

1 Image warping

1.1 Example code

We will be implementing an affine image warping function. The file `script.m` contains example code for how you might use the warping function, and some basic image loading and displaying. You should get something like the figure below.

Figure 1: See script.m.



1.2 Affine warp

An affine transform relates two sets of points:

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

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 2×2 matrix representing the linear component, and $\mathbf{t}$ is a 2×1 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:

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

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

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

Inputs: `im` is a grayscale `double` typed $height \times width \times 1$ matrix[1], `A` is a 3×3 matrix describing the transform ($\mathbf{p}^i_{warped} \equiv \mathbf{A}\mathbf{p}^i_{source}$), and `out_size=[out_height,out_width];` of the warped output image.
Outputs: `warp_im` of size `out_size(1)` $\times$ `out_size(2)` is the warped output image. The coordinates of the sampled output image points $\mathbf{p}^i_{warped}$ should be the rectangular range $(1, 1)$ to $(width, height)$ 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 `meshgrid` and `sub2ind`), although it might be easier to implement it using loops first.

Note: you can check your implementation to make sure it produces the same output as the following Matlab code (for grayscale or RGB images). Obviously the purpose of this exercise is practicing Matlab by implementing your own function without using `imtransform`.

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

```
function warp_im=warpA_check(im, A, out_size)
% warp_im=warpA(im, A, out_size)
% Warps (w,h,1) image im using affine (3,3) matrix A
% producing (out_size(1),out_size(2)) output image warp_im
% with warped  = A*input, warped spanning 1..out_size

% Remove last row (must be 0 0 1) and
% transpose because tform uses post-multiplication
A = A(1:2, :)';
tform = maketform( 'affine', A);
warp_im = imtransform( im, tform, 'nearest', ...
                     'XData', [1 out_size(2)], ...
                     'YData', [1 out_size(1)], 'Size', out_size );
```

1.3 Bilinear interpolation

You can also try to make a new function `warpAbilinear` exactly as above but using bilinear interpolation. Bilinear interpolation is the 2D equivalent of linear interpolation, and interpolates among the 4 closest pixels to $\mathbf{p}^i_{orig}$.

Let's have a quick ASCII look at linear interpolation in 1D first:

```
a      v?        b     <--- Values
o-------*---------o
i       p        i+1   <--- Positions
```

We want to sample the value `v` at position `p` (equivalent to $\mathbf{p}^i_{orig}$) which lies between integer positions `i` and `i+1`, with values `a` and `b` respectively. Note that `i=floor(p)`, and define `x=p-floor(p)`, so `v=a*(1-x)+b*x`. Bilinear interpolation can be implemented with three 1D interpolations (e.g., first interpolate along y the two closest columns and then interpolate across x the final row value). Again, this should be implemented with no `for` loops and you can check the result by substituting `'nearest'` with `'bilinear'`.