CS1114 Assignment 5, Part 1

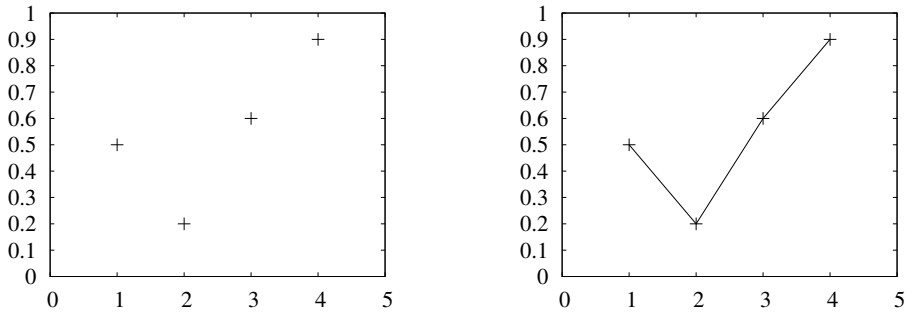
out: Friday, March 27, 2009.
due: **Friday, April 3, 2009, 5PM.**

This assignment covers three topics in two parts: interpolation and image transformations (Part 1), and feature-based image recognition (Part 2). This document contains Part 1. As usual, stub functions for the code you need to write can be found in `~/cs1114/student_files/A4`. Please copy these to your working directory.

1 Interpolation

Recall from lecture that interpolation involves figuring out a value at a point where you don't have data, but inside the range where you do. There are many techniques for interpolation, and we covered some of them in class. Let's start by considering linear interpolation in one dimension.

Suppose we have a (x, y) values with uniformly-spaced, integer x-coordinates, e.g. $(1, 0.5), (2, 0.2), (3, 0.6), (4, 0.9)$. These points, shown in Figure 1(a), might represent, for instance, the average stock price of Yoyodyne, Inc. for the first four days in January. As we discussed in lecture, linear interpolation fills in the in-between values by assuming a line segment connects each neighboring pair, as shown in Figure 1(b). Your first task is to write a function called `lerp`, which takes in a 1D vector (representing samples of a function) and an x value, and results the estimated function value at x using linear interpolation. Linear interpolation has three main steps: finding the two neighboring data points, computing the weights for the two points, then See the slides on interpolation for the formula for doing linear interpolation.



(a) Data points (b) Linear interpolation

Figure 1: Example of linear interpolation.

$\Longrightarrow$ Implement the `lerp` function in `lerp.m`.

Next, let's consider a 2D matrix of values at integer grid locations (e.g., a grayscale image). To interpolate values on a 2D grid, we can use the 2D analogue of linear interpolation: *bilinear interpolation*. In this case, there are *four* neighbors for each possible point we'd like to interpolation, and the intensity values of these four neighbors are all combined to compute the interpolated intensity, as shown in Figure 2. In

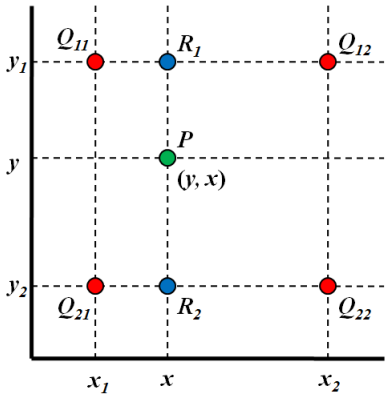


Figure 2: Bilinear interpolation.

the figure, the Q values represent intensities. To combine these intensitys, we perform linear interpolation in multiple directions: we first interpolate in the x direction (to get the value at the blue points), then in the y direction (to get the value at the green points). We can implement this by calling `lerp` three times.

$\Longrightarrow$ Implement bilinear interpolation in `bilerp.m`. This function takes in a matrix (an image, really), and an x, y value to compute the intensity at. You should return the computed intensity. This function must call your `lerp` function.

2 Image transformations

First, a note on images: remember that Matlab has different data types for storing an image (e.g., *double*, *uint8*, and *logical* (for binary images)). The range of intensities for each type is different (e.g., a `double` image has intensities in the range $[0, 1]$ and a `uint8` image has intensities in the range $[0, 255]$).

One other thing you should keep in mind when you're working with images in this assignment is that, as you've found, you access a value of a matrix by giving the row, then the column. For a matrix representing an image, this is counter-intuitive, as in 2D coordinate systems the x-value (column) usually precedes the y-value. This is a source of headaches, but is difficult to get around. In addition the y-axis of an image in Matlab is inverted. Matlab indexes the image like a matrix, so increasing values of i move you down in the image, rather than up. This has some slightly annoying consequences when applying geometric transformations on images.

Next, we'll use these interpolation functions to help us implement image transformations. Recall that a 2D affine transformation can be represented with a 3×3 matrix T:

$$T = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix}$$

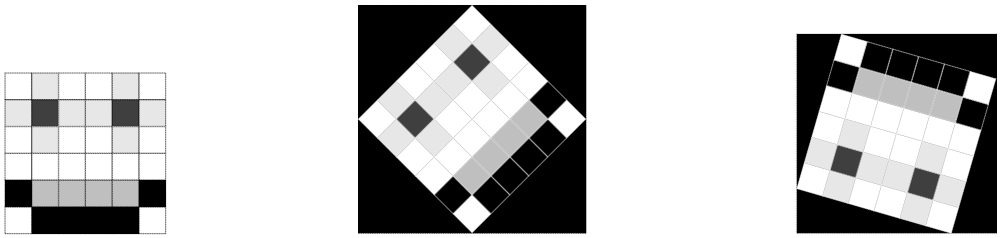


Figure 3: The dimensions of a rotated image may vary from the those of the original.

Recall that the reason why this matrix is 3×3, rather than 2×2, is that we operate in *homogeneous* coordinates; that is, we add an extra 1 on the end of our 2D coordinates (i.e., (x, y) becomes $(x, y, 1)$), in order to represent translations with a matrix multiplication. To apply a transformation T to a pixel, we multiply T by the pixel's location:

$$T \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} ax + by + c \\ dx + ey + f \\ 1 \end{bmatrix}$$

In Matlab, matrix multiply is done with the normal $*$ operator. Note that the number of columns in the 1st matrix must equal the number of rows in the 2nd matrix.

To apply a transformation T to an entire image I, we could apply the transformation to each of I's pixels to map them to the output image. However, this *forward warping* procedure has several problems, as mentioned in class. Instead, we'll use *inverse mapping* to warp the pixels of the output image back to the input image. Because this won't necessarily hit an integer-valued location, we'll need to use interpolation to determine the intensity of the input image at the desired location.

One other issue is that the transformed image might not fit into an image of the same size as the input image, as shown in Figure 3. Thus, you will need to determine the correct size for the output image.

$\Longrightarrow$ Write a function `transform_image` that takes as input a grayscale image (in `double` format) and a 3×3 affine transform T. This function will return the transformed image. This function should use inverse warping, and will call your `bilerp` function. We have provided an image `duck_gray.png` included in the `A5` directory for testing.

Next, you will write a function to transform RGB images. To do this, you will simply call `transform_image` three times, once for each channel, then put the results together into a single image. You can use the `image_rgb` function to break an RGB image up into three channels, e.g.:

```
>> img = imread('duck_rgb.png');
>> [R, G, B] = image_rgb(im2double(img));
```

To assemble the RGB channels back into a single image, you can use the Matlab colon

operator.

$\Longrightarrow$ Write a function `transform_image_rgb` that takes as input an RGB image (in `double` format), and a 3×3 affine transform T, and returns the transformed RGB image. You must call `transform_image` from this function. You might also find the `inv` function handy. You can use the image `duck_rgb.png` to test your function.

To demo your transformation function, please write a function called `demo_transform` that calls your `transform_image_rgb` with several different transformations, and displays the results. Your function should at least demo:

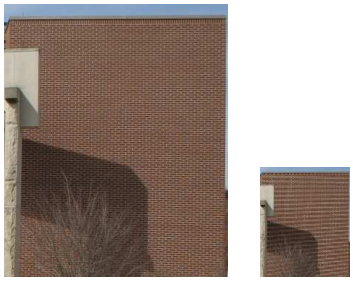
1. Horizontal flipping
2. Scaling by a factor of 0.5
3. Rotation by 45 degrees around the center of the image

but you may include any others that you like. You will need to figure out the right transformation matrices to accomplish these tasks.

One simple way to show multiple images at a time is to show them in separate `figure` windows. You can use the `figure` command to accomplish this, bearing in mind that `imshow` draws an image in the most recently opened figure window (or creates one of its own if none is open). Please show the original image, as well as the transformed images. You may use the image `duck_rgb.png`, or use an image of your own.

Note that this function can take a long time to run. One reason is that function calls take a *long* time to execute in Matlab. We will assign one point of extra credit if you can get your *transform_image_rgb* to be at least twice as fast as our implementation.

4 Anti-aliasing



(a) (b)

Figure 4: Example of aliasing. (a) Original image and (b) downsampled image with aliasing artifacts.

There is a problem with our interpolation method above: it is not very good at shrinking images, due to aliasing. For instance, if you try to downsample the image `bricks_rgb.png` by a factor of 0.4, you get the image shown on the left of Figure 4; notice the strange

banding effects in this image. The problem is that a single pixel in the output image corresponds to about 2.8 pixels in the input image, but we are sampling the value of a single pixel—we should really be averaging over a small area. To overcome this problem, we will create a data structure that will let us (approximately) average over *any* possible square regions of pixels in the input image: an *image stack*. An image stack is a 3D matrix that you can think of as, not surprisingly, a stack of images, one on top of the other. The top image in the cube will be the original input image. Images further down the stack will be the input image with progressively larger amounts of blur. The size of the matrix will be `rows` $\times$ `cols` $\times$ `num_levels`, where the original (grayscale) image has size `rows` $\times$ `cols` and there are `num_levels` images in the stack. Figure 5 shows a few images from an example stack.

(a) Level 1 (b) Level 5 (c) Level 20

Figure 5: Three slices from an image stack.

Before we use the stack, we must write a function to create it. You will first write a function `create_image_stack`, which takes as input a (grayscale) image in double format and a number of levels in the stack, and returns a 3D matrix `stack` corresponding to the stack. (Note that this is a different type of stacks than the stack from A3.) Again, the first image on the stack, i.e. `stack(:,:,1)` will be the original image. Every other image in the stack will be a blurred version of the previous image. You may use the `conv2` Matlab function to do the blurring. A good blur kernel to use is:

$$K = \frac{1}{9} \cdot \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

(If you come up with a better one, please let us know.) Now, for image k in the stack, we know that every pixel is a (weighted) average of some number of pixels (a $k \times k$ patch, roughly speaking) in the input image. Thus, if we downsample the image by a factor of k, we want to sample pixels from level k of the stack.

$\Longrightarrow$ Write a function `create_image_stack` that takes a grayscale image and a number `max_levels`, and returns an image stack.

Now, what happens if downsample the image by a fractional factor, such as 3.6? Unfortunately, there is no level 3.6 of the stack. Fortunately, we have a tool to solve this problem: interpolation. We now potentially need to sample a value at position (row, col, k) of the image stack, where all three coordinates are fractional. We therefore something more powerful than bilinear interpolation: trilinear interpolation! Each position we want to sample now has eight neighbors, and we'll combine all

of their values together in a weighted sum. This sounds complicated, but we can write this in terms of our existing functions. In particular, we now interpolate separately along different dimensions: trilinear interpolation can be implemented with two calls to `bilerp` and one call to `lerp`.

$\Longrightarrow$ Write a function `trilerp` that takes an image stack, and a row, column, and stack level k, and returns the interpolated value.

Now we can finally write a transformation function that does proper *anti-aliasing*. In order to do this, create a function `image_transform_aa`. This function will be almost identical to `image_transform_aa`, except for three main changes. The first change is to compute the image stack. The second change is to compute, for the transformation T, how much T is scaling down the image. If T is defined by the six values a, b, c, d, e, f above, then, to a first approximation, the downscale factor is:

$$k = \frac{2}{\sqrt{a^2 + b^2} + \sqrt{d^2 + e^2}}$$

However, if $k < 1$ (corresponding to scaling *up* the image), we still want to sample from level 1. This situation reverts to normal bilinear interpolation.

The final change we need to make is to call the `trilerp` function on the image stack, instead of `bilerp` on the input image.

$\Longrightarrow$ Create `transform_image_aa`. You should also create a corresponding function `transform_image_rgb_aa`.

$\Longrightarrow$ Finally, create a function `demo_transform_aa` which shows the results of downsampling an image using bilinear versus trilinear interpolation. You may use an image we provide, or any other image of your choice, as long as it demonstrates the difference in an obvious way.

5 What to turn in

To recap, you will turn in the following functions for Part 1:

1. `lerp` (Section 2)
2. `bilerp` (Section 2)
3. `transform_image` (Section 3)
4. `transform_image_rgb` (Section 3)
5. `demo_transform` (Section 3)
6. `create_image_stack` (Section 4)
7. `trilerp` (Section 4)
8. `transform_image_aa` (Section 4)
9. `transform_image_rgb_aa` (Section 4)
10. `demo_transform_aa` (Section 4)