Problem Set 0
Assigned: 02/03/2004
Due: 02/12/2004

Introduction

This assignment is an introduction to MATLAB's image processing toolbox. It assumes that the reader is already acquainted with MATLAB and has general knowledge of how images are represented and stored by computer programs. In this section we provide a brief discussion of how images are treated in MATLAB. In the following section we list a few problems that will give you some experience using this toolbox. Those already familiar with using images in MATLAB can go directly to the next section.

MATLAB stores images as an nD-array of dimensions $h \times w \times d$ and class `uint8`, where h, w, and d are the height, width, and channel count of the image (grayscale images have one channel and color images have three). Images can be read and written using various image compression formats using MATLAB's `imread` and `imwrite` functions. Examples of popular, supported image formats are JPEG, windows bitmap, TIFF, and PNG. Images are displayed in MATLAB using `imshow`. The function `imagesc` can also be used as explained later. Below is an example snippet of MATLAB code that demonstrates the use of the above functions. As seen below, this code reads a TIFF image, displays it in a figure and then saves it in JPEG format with 75 percent quality.

```
...
im = imread('myimage.tif');
imshow(im);
imwrite(im, 'myimage.jpg', 'JPG', 'Quality', 75);
...
```

Arithmetic operators in MATLAB can only operate on data of class `double`. Therefore images must be converted to double format prior to being manipulated in MATLAB when using functions or arithmetic operators outside of the image processing toolbox (note some functions in the toolbox, like `hsv2rgb` also require double format). Images can be converted to class `double` using the function `im2double`. It is common that after processing an image, its values no longer vary from 0 to 255. In fact, an image may be

purposefully scaled such that its values vary in some pre-defined range (e.g. between 0 and 1). Such images can be quickly visualized using `imagesc`, which scales such image data to use the full colormap.

The image processing toolbox offers a rich set of functions for manipulating and processing images. All the functions included by your version of the image toolbox can be viewed by typing '`help images`' from the MATLAB command prompt. The following problems will provide you experience with using the toolbox. Over the course of the semester you will become familiar with many of its functions.

Problems

1. **Image correction for an image digitizer.**

   Load in and display the image `original.tif`. This is the (synthesized) output from a hypothetical 35mm slide scanner. The slide to be digitized is illuminated with a lamp and imaged onto a plane in space. A 1-d detector array is moved through that image plane and records the amount of light falling on each pixel as it scans through the image. original.tif is the raw output of a scan. Each vertical column represents the output of a single pixel of the detector. Each horizontal row represents a di.erent time step of the detector through the image.

   Notice the vertical streaks through the image. These are because not every element of the detector array has equal sensitivity to light. Since the 1-d detector array sweeps through the image vertically, the less sensitive array elements leave dark streaks through the image.

   You remove the slide from the scanner, and record a calibration white field image (`whiteField.tif`). This image shows the vertical streaks, and also unevenness in the illumination of the slide area. Both these multiplicative e.ects can be compensated for by dividing the slide image by the white field image (point-by-point division, using ./, not the matrix division operator, /). In other words, at every pixel position, divide the intensity of each color of the original image by the intensity of the white .eld image at that position. That will normalize for the di.erences in sensitivity and illumination. Display the resulting corrected image.

   Solid-state detectors typically have a linear response to light intensity, while monitor displays typically have a non-linear response, often modeled by an exponential non-linearity. Find a compensating nonlinearity

which leads to a pleasing tonescale for the displayed image by raising each pixel to some power in the range of 0.3 to 0.5. Be aware of the overall image scale.

2. **CCD color interpolation.**

   To record color images with the single-chip CCD cameras typically used in digital cameras, arrays of color filters are used. Each pixel is covered by a filter of some color, giving a sample of only one color at each spatial position on the detector (see slides in ccd.ppt). Digital cameras need to reconstruct an image with 3 color samples at each position from the recorded data of just one color sample per position. (The human eye needs to do this, too.)

   Here we will show one simple interpolation method, to practice image array manipulations in matlab. Load in the 3-color image `brettan.tif`.

   (a) Simulate the sampling by the CCD. Assume that the color filter pattern is R, G, B stripes, in a repeating pattern. Display a black and white image showing the intensity of each pixel in the color band of the color filter stripe that it lies under.

   (b) Linear interpolation. Form a color image from the simulated data of part (a) by linearly interpolating between nearby color samples in the same row. For example, if the pixel at position (1,1) is a red sample, then the red value at position (1,2) should be

   $$\hat{I}(1,2) = \frac{2}{3}I(1,1) + \frac{1}{3}I(1,4)$$

   where I is the array of sampled intensities from (a). Display the resulting full-color picture.

   (c) The undersampling causes color fringe artifacts. To reduce those, the image can be optically blurred before falling on the CCD. Load in the blurred image, `brettanBlurred.tif`, and perform the same steps as in (a) and (b). Are the color fringes reduced? At what price? Later in the term, we will discuss a non-linear color sample interpolation method that requires much less image blurring in order to remove color fringe artifacts.

3. **Object metamorphosis.**

   Object metamorphosis is an important phenomena in nature. The ability to morph between images of an object or multiple objects is a useful tool in both vision and graphics. Below we will explore some of MATLAB's image interpolation capabilities with an example image morph.

   The images `neutral.jpg` and `smile.jpg` depict an actor with a neutral and smiling facial expression. The files `flowNS.mat` and `flowSN.mat` are 2-D optical flow fields, which map the location of each pixel in one image to a corresponding pixel in the other (`flowNS.mat` is a flow field from neutral to smiling and `flowSN.mat` is from smiling to neutral). Below we synthesize a smooth transition from neutral to smiling using image morphing. There are two basic steps to image morphing: (1) each image is aligned to have the same shape, (2) the aligned images are blended to form the morphed image. The first step, known as image warping, is performed as follows,

   $$I_w(x, y) = I(x + dx, y + dy)$$

   where I_w is the warped image, I is the input image, and (dx, dy) is the optical flow vector at location (x, y) of the other input image (this is referred to as *reverse* warping). The shape of one image can be transitioned to that of the other by moving along the defined flow vectors. For example, each image can be warped to their midpoint combined shape by moving 0.5 along each flow field. The blended images form an image that is often referred to as the *average* image. Using MATLAB's `interp2` function and the provided flow fields, warp each image to the average shape by first multiplying each flow field by 0.5 and then warping each image using their respective flow fields. Make sure to avoid or remove any `NaN`'s introduced by `interp2`.

   A smooth transition between images is generated using a blending parameter α, where $0 \leq \alpha \leq 1$. Given two input images I_1 and I_2 we morph them to an intermediate transition step α as follows,

   $$\begin{aligned} I_w^1(x, y) &= I_1(x + \alpha dx, y + \alpha dy) \\ I_w^2(x, y) &= I_2(x + (1 - \alpha)dx, y + (1 - \alpha)dy) \\ I_m(x, y) &= (1 - \alpha)I_w^1(x, y) + \alpha I_w^2(x, y) \end{aligned}$$

where I_m is the morphed image. Compute and display the average image. Compute a smooth transition between the two images by varying α on the interval $(0, 1]$ in 0.1 increments. Display each transition step in a 2×5 subplot.

4. **Simple background subtraction using white screening.**

   In this problem we explore a simple, useful tool in the movie special effects industry known as white screening. We will spice up the results from the previous problem by adding an original background to the morph sequence. In doing this problem you will gain some experience in performing simple image filtering and morphological operations in MATLAB.

   The input images from the previous problem exhibit a whitish background. Ideally we would like the background to be a solid color (not present on the actor) that we could use to identify it. Unfortunately, the background wall exhibits various texture and lighting effects. Additionally their is noise in the image formation process which makes it almost impossible to capture a solid color in the background. To remove the background, we will therefore assume that the range of colors present all lie within a 3-D Gaussian in the RGB color space. To compute the mean and covariance of this Gaussian, select two points in the upper-left corner of either image that outline a large rectangular region of the background using MATLAB's `ginput` function. Compute the mean and covariance of the pixels in the selected rectangular region.

   To distinguish points that are within the support of the multivariate Gaussian we will use the Mahalanobis distance,

   $$d = (\mathbf{x} - \mu_{\mathbf{x}})^T \mathbf{C}^{-1} (\mathbf{x} - \mu_{\mathbf{x}})$$

   where $\mu_{\mathbf{x}}$, $\mathbf{C}$ are the mean and covariance of the Gaussian and $\mathbf{x}$ is a pixel in the image. Background subtract each image by computing the Mahalanobis distance for each pixel, discarding pixels whose distance is under some threshold. Adjust the threshold as you see fit, until the background is properly segmented. We found a threshold of 200 to suffice. Mark the background with black (i.e. $(0, 0, 0)$) in each image. Try not to use `for` loops as this will make the routine quite slow in MATLAB.

To cleanup your background segmentation result, first use a median filter to remove spurious pixels from your background (if any). You can do this using MATLAB's `medfilt2` function. After background subtraction you will notice an annoying white silloette around the actor. Remove it by using MATLAB's `imerode` function. We found a `line` structured element with length 11 and a 90 degree orientation to work well.

Next, we will generate a small video sequence with the processed images: As in the previous problem, compute a transition between each of the background segmented images. Replace the background in each image with the image `beach.jpg`. (Hint: You can do this by computing image masks.) Finally, composite all the images into a movie sequence using `images2movie` and display it with `showmovie`. These routines are included with this problem set and are available from the course website.