Common MATLAB pitfalls

MATLAB is an excellent tool for implementing image processing algorithms quickly, but there are some issues that can leave you scratching your head, or worse, unknowingly producing incorrect results. The purpose of this document is to highlight some of these issues and make your work on the MATLAB homework more pleasant.

1 Integer and Floating Point Images and Image Display

1.1 Integer-valued images

There are many different image formats, but most use one or more unsigned integers to represent each pixel. Unsigned integers can only represent values between 0 and $2^K - 1$, where $K$ is the number of bits used to represent the integers. What happens to them when you exceed this range depends on the programming language you use. Understanding this issue is key to getting correct results when using MATLAB for image processing.

The `imread` function of MATLAB returns different image value types depending on the images used. In our class, the majority of the provided images are grayscale JPEG images specified by a single unsigned 8-bit integer (uint8 type) at each pixel, which ranges between 0 and 255. If a uint8 image value of 255 is increased by some amount, then it is clipped to 255 (overflow problem), whereas, if a uint8 0 value is decreased by some amount, then it is clipped to 0 (underflow problem). For example:

```matlab
>> uint8(255) + 1
ans = 255
>> uint8(0) - 1
ans = 0
```

When applied to an entire image, the previous overflow and underflow problems can lead to surprising and unwanted results. For example, see Fig. 1. This problem can be avoided by transforming the unit8 values of an image to single or double precision floating point values by using the `single` or `double` functions of MATLAB, as it is shown in Fig. 1.

1.2 Floating point images

It is frequently convenient to convert an image to a floating-point type using the MATLAB function `double`. However, many MATLAB functions, such as `imshow`, `imnoise`, `imfilter`, `imhist`, and other less common functions, do not work properly if the input image is “double”; see Fig. 2 for an example. As a rule of thumb, you will get correct results by converting any type of image, including a “double” image, using the MATLAB function `mat2gray` before using it for further display and analysis. You can do that when you read the image at the beginning of your code by writing `im = mat2gray(imread(...))`.

1.3 Image display

The MATLAB function `imshow` is the most frequently used and abused function for image display. By default, the range of displayed values depends on the data type of the image to be displayed. So, a `unit8`
Figure 1: Top: original uint8 image “im”. Middle left: all values of $\text{im-128}$ that are below zero are clipped to 0 and that is why $\text{im-128+128}$ looks strange. Middle right: all values of $\text{im+128}$ that are above 255 are clipped to 255 and that is why $\text{im+128-128}$ does not look the same as $\text{im}$. Bottom: Using $\text{double(im)}$ instead of $\text{im}$ avoids the previous overflow and underflow problems.
Figure 2: Note that, although `imshow(im)` displays the image correctly, this is not true when using `double(im)`. However, this is not a problem when `double(im)` is converted by `mat2gray`. 
image \textit{im} taking integer values from 0 to 255, is mapped by \textit{imshow} so that zero-valued pixels are black and 255-valued pixels are white. Values between 0 and 255 are displayed as intermediate shades of gray. When using image \textit{double(im)}, \textit{imshow} maps zero-valued pixels to black and pixels with values 1 or above to white. On the other hand, when using image \textit{mat2gray(im)}, which always takes values between 0 and 1, \textit{imshow} maps zero-valued pixels to black and pixels with values 1 to white, where values between 0 and 1 are displayed as intermediate shades of gray. Note however that \textit{mat2gray(im)} converts the values of \textit{im} to be between 0 and 1 so that the minimum value of \textit{im} is mapped to 0 and the maximum value of \textit{im} is mapped to 1. Examples as shown in the first column of Fig. 3. Note finally that the range of displayed values can also be specified explicitly by using \textit{imshow(im,[low high])}, as shown in the second column of Fig. 3. The \textit{imshow} function displays the value low (and any value less than low) as black, and the value high (and any value greater than high) as white. Values between low and high are displayed as intermediate shades of gray. If an empty matrix ([ ]) is specified, \textit{imshow} uses [\text{min}(\text{im}(:)) \text{max}(\text{im}(:))] for [low high]. In other words, it uses the minimum value as black, and the maximum value as white.

2 Frequency Domain Operations

2.1 Displaying the magnitude Fourier transform

Several MATLAB homework problems in this class are performed in the frequency domain using the Fourier transform. The preferred way to visualize the magnitude of the Fourier transform of an image is to place low frequencies near the center and high frequencies near the corners of the image plane. However, the function \textit{fft2} of MATLAB does exactly the opposite: it places low frequencies near the corners and high frequencies near the center. To address this issue, MATLAB provides two functions: \textit{fftshift} and \textit{ifftshift}. Figure 4 shows examples of a correctly displaying the magnitude Fourier transform, which requires log-scaling and appropriate shifting.

2.2 Shifting in the Fourier domain

A common error when using \textit{fftshift} in the frequency domain is that careless use can produce complex-valued images when converted back to the spatial domain. Figure 5 shows that, if an image is shifted in the frequency domain, then it produces an odd result when it is inverted back to the space domain. This is due to the fact that a shift in the frequency domain introduces complex-valued scaling in the spatial domain. To address this issue, you should either remove the shift in the frequency domain before inverting by using \textit{ifftshift} or take the magnitude of the resulting image in the space domain.

Note also that you must be careful how you use \textit{fftshift} and \textit{ifftshift}, since they may not do the same job depending on whether the image dimensions are both even or at least on of them is odd. If both dimensions of an image \textit{im} are even, then \textit{fftshift(fft2(im))} = \textit{ifftshift(fft2(im))}. However, if at least one of the dimensions is odd, then \textit{im} are even, then \textit{fftshift(fft2(im))} ≠ \textit{ifftshift(fft2(im))}.

2.3 Filtering and shifting

When filtering an image in the frequency domain, it is important that you shift the filter and the transformed image the same amount. Figure 6 shows the results of blurring an image when the shift is the same and when it is not.

2.4 Using correct FFT function

Another simple, but surprisingly common error, is using \textit{fft} instead of \textit{fft2} to compute the Fourier transform of a 2D image. Using \textit{fft} will perform a 1D FFT on each column of the image. The same
Figure 3: `imshow` results for different image types and ranges. Note that by specifying an explicit range allows the image `double(im)` to be displayed correctly. Guess which of the results are identical.
Figure 4: The middle left is the most common way to correctly display that magnitude of the Fourier transform of an image by shifting and log-scaling. However, the middle right is also an acceptable way, provided that you keep in mind that the low frequencies are located near the corners of the image plane. Note that the DC component (at zero frequency) of the Fourier magnitude is often much higher than all other values. This necessitates using log-scaling, as most visual information is lost otherwise; see lower left. Note also that improper use of `imshow` can result in a bad display of the result; see lower right.
Figure 5: Shifting an image (top left) in the Fourier domain and then inverting it back in the space domain introduces a complex-valued scaling factor. As a consequence, the resulting image is not displayed properly by `imshow` (top right). To address this problem, you can either remove the shift in the frequency domain before inverting (bottom left), or take the absolute value of the resulting image before displaying it (bottom right).
Figure 6: Since the Fourier transform of the input image (top right) is shifted (top left), it is important that the frequency response of the blurring filter is also shifted (middle left) in order to obtain the right result (middle right). The bottom row shows what will happen when using an incorrectly shifted frequency response.
Figure 7: Applying the function \texttt{fft} once on an input 2D image (top left) does not produce the Fourier transform of the image (top right). This transform can be obtained by using the function \texttt{fft2} (bottom left). It can also be obtained by using \texttt{fft} twice, once on the columns of the input image and once on the rows of the result. Note the use of function \texttt{imrotate} that rotates the result 90 degrees counterclockwise.

Problem occurs when using \texttt{ifft} instead of \texttt{ifft2}. This is illustrated in Fig. 7. Note that the 2D Fourier transform can be also be obtained by using \texttt{fft} twice, once on the columns of the input image and once on the rows of the result.