Assignment 1
Signal & Image Processing

Casper B. Hansen
University of Copenhagen
Department of Computer Science
fvx507@alumni.ku.dk

September 4, 2014

1 1.1 Image generation and manipulation of pixels

In this exercise we are to generate a 20x20 image, show it and take cursor input, which is then used to manipulate the pixel at that location.

```
%% assignment 1.1

figure(1);                                  % start figure 1
img = rand(20);                             % generate greyscale image
imshow(img), imagesc(img);                  % show and scale the image
axis image, xlabel('X'); ylabel('Y');       % set x and y axis labels

[y, x] = ginput(1);                         % take cursor input
img(uint8(x), uint8(y)) = 0;                % read position, set it to black

imshow(img), imagesc(img);                  % show and scale the image
axis image, xlabel('X'); ylabel('Y');       % set x and y axis labels
clear all;                                  % fixes title rewrite bug
```

Figure 1: Excerpt showing the solution of 1.1 (assignments.m)

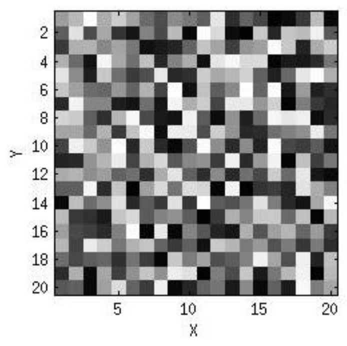

Figure 2: Before; random generated image

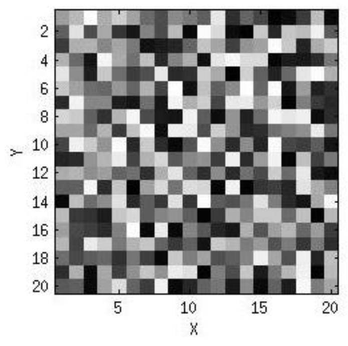

Figure 3: After; cursor input at $(13, 5)$

As shown in figure (3), the program work as intended. In the figures above, the change is evident at pixel indices $(13, 5)$.

2 1.2 Display methods

We are to examine the differences in MatLab's methods of displaying data visually. From *Signal Processing Toolbox* we have `imshow`, which constrains the proportions of visualizations.

```
%% assignment 1.2

img = imread('images/cell.tif');

subplot(1,3,1), imshow(img), title('imshow');
subplot(1,3,2), image(img), title('image');
subplot(1,3,3), imagesc(img), title('imagesc');
colormap(jet);
```

Figure 4: Excerpt showing the solution of 1.2 (assignments.m)

Unlike `imshow`, the generic `image`, and quite confusingly, doesn't interpret the data as an image in the sense of a picture, but rather maps the matrix data pictorially, and thus does not require proportional constraints.

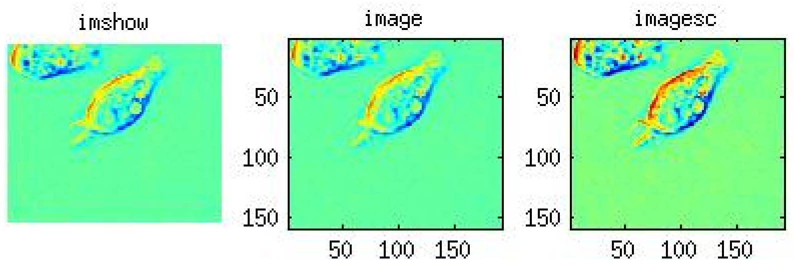

Figure 5: Differences in image display functions

Lastly, the `imagesc` takes the dynamic range of the data, and scales it to the extremities (minimium and maximum). It allows us to see details that weren't perceptible beforehand.

3 1.3 Bit-slicing

We are to apply the bit-slicing technique on an arbitrary test image. For this I've chosen the `images/cameraman.tif` picture.

```
%% assignment 1.3

img = imread('images/cameraman.tif');    % read in grey-scale image

for bit = 1:8                                % loop through bits 1-8
    plane = bitget(img, bit);                % get bit plane at current
    subplot(2, 4, bit), imshow(plane, []);   % subplot current bit plane
    title(strcat('Bit', num2str(bit)))       % add title
end
```

Figure 6: Excerpt showing the solution of 1.3 (assignments.m)

As we can see, in figure (7), the first 3 least significant bits doesn't contribute much to the picture — arguably not the fourth either. From bit 5 and on, we gradually observe an increase in being able to descern objects within the scene, or picture.

Bits 6 and 8 have the highest perceptability in this example.

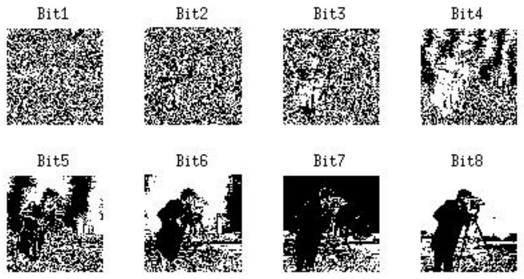

Figure 7: Bit-slicing applied to `images/cameraman.tif`

4 **1.4** RGB to HSV conversion

For this assignment, we are to convert an RGB image into HSV colormapping, and display the changes, as well as its individual channel components.

```matlab
%% assignment 1.4

img = imread('Profile.jpg');     % read in RGB image
hsv = rgb2hsv(img);
components = ['H', 'S', 'V'];    % define component array

subplot(1,5,1), imshow(img); axis image; title('RGB');   % plot RGB
subplot(1,5,2), imshow(hsv); axis image; title('HSV');   % plot HSV
for c = 1:3
    subplot(1,5,c+2), imshow(hsv(:,:,c));                % plot component
    title(components(c));                                % component title
end
```

Figure 8: Excerpt showing the solution of 1.4 (assignments.m)

In the figure (9) below, we see a clear difference in appearance between the two formats. The HSV colormap allows us to see details in a picture much clearer, as it separates much of the visual informations out into separate channels.

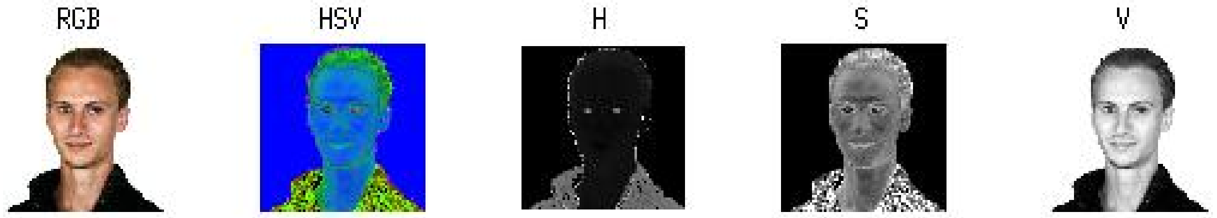

Figure 9: HSV converted from RGB, including individual channels

5 **1.5** Spatial resolution

We are to examine how the digitization process affects the outcome of the process.

```
%% assignment 1.5

img = imread('Profile.jpg');          % read in RGB image

figure (1);       % figure 1, scaling
low = imresize(img, 0.1);             % scale by 0.08
squeeze = imresize(img, [64 128]);    % squeezed y-axis
upscale = imresize(low, 10);          % upscaled

subplot(1,3,1), imshow(low); axis image; title('Low Resolution');
subplot(1,3,2), imshow(squeeze); axis image; title('Squeezed');
subplot(1,3,3), imshow(upscale); axis image; title('Up-scaled');

figure (2);      % figure 1, interpolation
tic, nearest  = imresize(low, 10, 'nearest'); toc    % 6.830 ms
tic, bilinear = imresize(low, 10, 'bilinear'); toc   % 21.617 ms
tic, bicubic  = imresize(low, 10, 'bicubic'); toc    % 40.825 ms

subplot(1,3,1), imshow(nearest); axis image; title('Nearest');
subplot(1,3,2), imshow(bilinear); axis image; title('Bilinear');
subplot(1,3,3), imshow(bicubic); axis image; title('Bicubic');
```

Figure 10: Excerpt showing the solution of 1.5 (assignments.m)

In figure (10) I present three resizing alterations to an image; a low-resolution image, a squeezed image (unproportionally scaled), and lastly an up-scaled image.

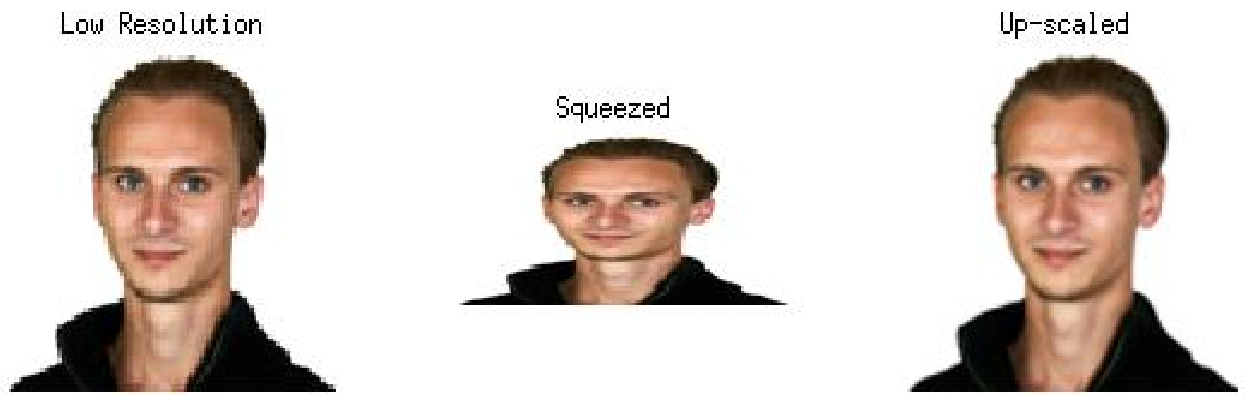

Figure 11: Image alterations that induces artifacts

The loss of detail is quite evident in the low-resolution image (an tenth the size of the original), and thus appears very *pixelated*. The squeezed image loses much of the vertical detail information, and if stretched back into proportion would appear somewhat blurred on the vertical axis. Lastly, the up-scaled image attempts to recreate detail from lost information. In other words, it must invent pixels by looking at nearby pixels.

To further expand on the notion of *inventing pixels*, we examine three algorithmic techniques for doing so; nearest, bilinear and bicubic.

Each of these algorithms come at an expense to the computation time required to perform the operation. Therefore their practical application depends on how fast one must compute the result (ie. real-time applications should generally be very fast).

Nearest 6.830 ms **Bilinear** 21.617 ms **Bicubic** 40.825 ms

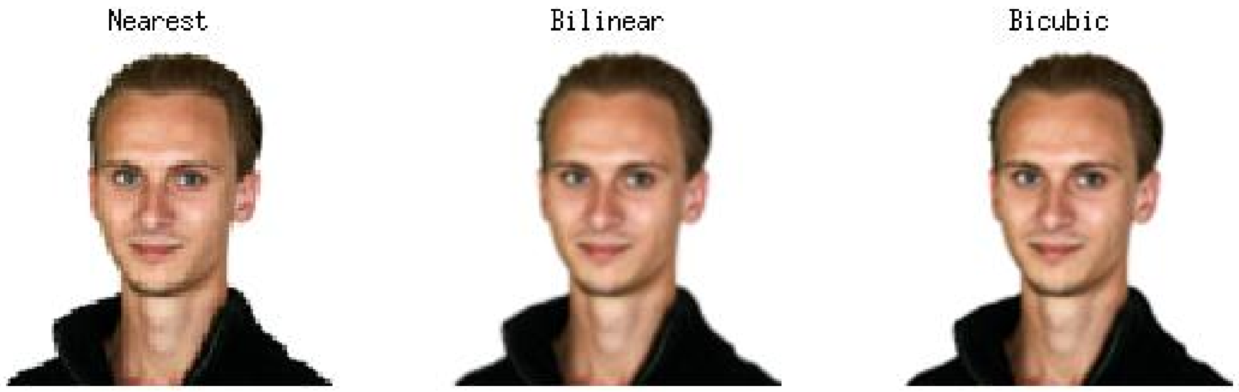

Figure 12: interpolation algorithms applied to a low-resolution image

As we can see, the nearest algorithm is by far the fastest, but also produces the worst result — much detail remains lost, and features are obscurred. The bilinear comes in second and produces a far better image, where most of the features are preserved well. The last and best, but also the slowest, is the bicubic algorithm, which preserves much detail and does a very good job at recreating the original image.

6 1.6 ...

I didn't have time to do this one :(

7 **1.7** arithmetic operations

We are to examine how arithmetic operations affect the outcome of images. Also, the assignment asks us to examine resizing of the images to be used, such that they are of the same size. I found this difficult to understand however, since the images are already of the same size.

```

%% assignment 1.7

A = imread('images/rice.png');
B = imread('images/cameraman.tif');
C = imadd(A, B); D = imadd(A, B, 'uint16');
E = imsubtract(A, B);

subplot(1,3,1), imagesc(C), axis image, title('imadd 8-bit');
subplot(1,3,2), imagesc(D), axis image, title('imadd 16-bit');
```

Figure 13: Excerpt showing the solution of 1.7 (assignments.m)

The + and − operators are, as far as I could tell from the documentation, equivalent to `imadd` and `imsubtract` in how they work on basic appliance — only special cases give rise to differences, so one set will be omitted.

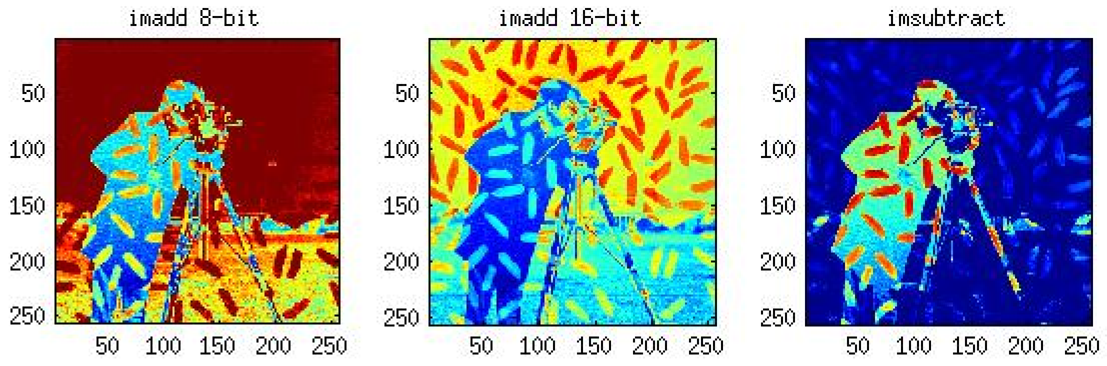

Figure 14: Outcomes of arithmetic operations

As we can see from figure (14), there is a clear difference in how much detail is produced by the 16-bit addition operation, compared to that of the 8-bit addition — one can clearly descern both pictures in the 16-bit version.

8 1.8 Differences across sequences

We use the `imabsdiff` function to determine the difference between two images. Doing so over a sequence of n images produces $n - 1$ images.

```

%% assignment 1.8

A = cell(1,10);
B = cell(1,9);

for i = 1:10
    if i < 10
        A{i} = imread(strcat('images/AT3_1m4_0', num2str(i), '.tif'))
    else
        A{i} = imread(strcat('images/AT3_1m4_', num2str(i), '.tif'))
    end
end

for i = 1:9
    img = imabsdiff(A{i}, A{i+1});
    subplot(2,5,i), imagesc(img), axis image;
```

Figure 15: Excerpt showing the solution of 1.8 (assignments.m)

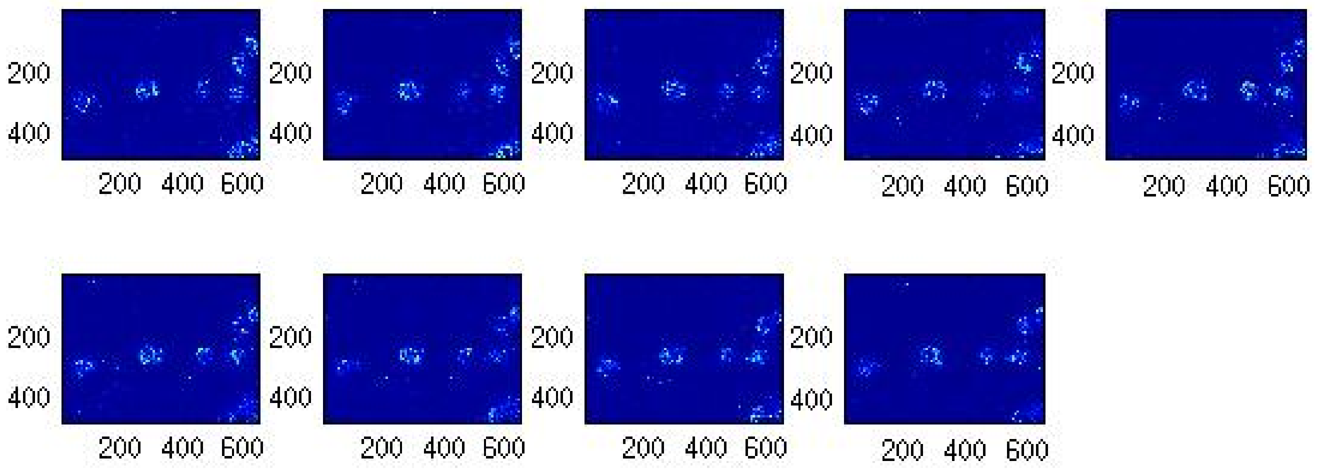

Figure 16: Differences over a sequence of 10 images

9 1.9 Writing functions in MatLab

```matlab
function [ O ] = blend(A, B, w_A, w_B)
    O = (A.*w_A) + (B.*w_B);
end
```

Figure 17: Excerpt showing the solution of 1.9 (blend.m)

Writing a function must occur in its own file (here `blend.m`), where I declare it as taking 4 arguments; the two images, and the two weight coefficients. It performs the weighting element-wise on both images and adds their respective results to form the returned image.

In the code figure below, an example of its application is given.

```matlab

%% assignment 1.9

A = imread('images/rice.png');
B = imread('images/cameraman.tif');
C = blend(A, B, 0.25, 0.75);
```

Figure 18: Excerpt showing the application of the solution for 1.9 (assignments.m)

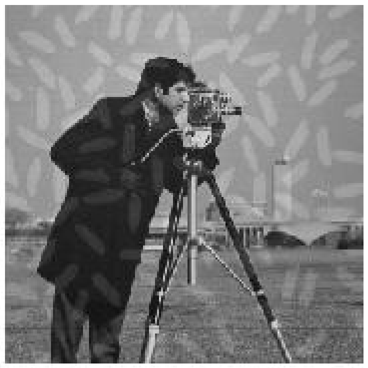

Figure 19: Shows the result of code figure (18)