Jonah Rubino, Yoni Carver
Assignment #3

Part 1 – Least Significant Bit Data Hiding

When the image "peppers.tif" is placed through the function that determines and shows the bit planes of an image, the highest bit plane you can observe that no longer represents the image is bit plane 3. When the image "baboon.tif" is placed through the same function, the highest bit plane that just resembles noise, is bit plane 4. These two bit planes are different for these images. This is likely due to the peppers image having more information within the lower bit planes. The baboon's lack of information comparatively, causes it to look like noise in a higher plane.

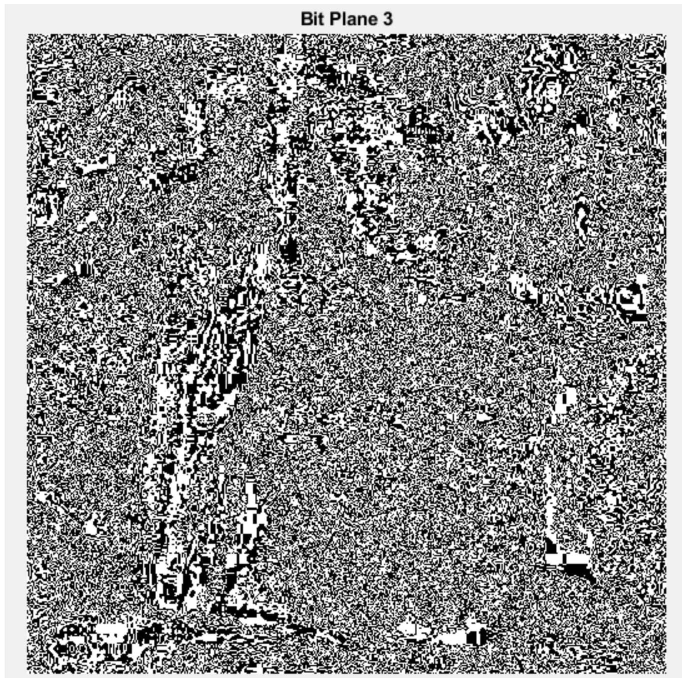

Figure 1: Bit plane 3 of the original peppers image

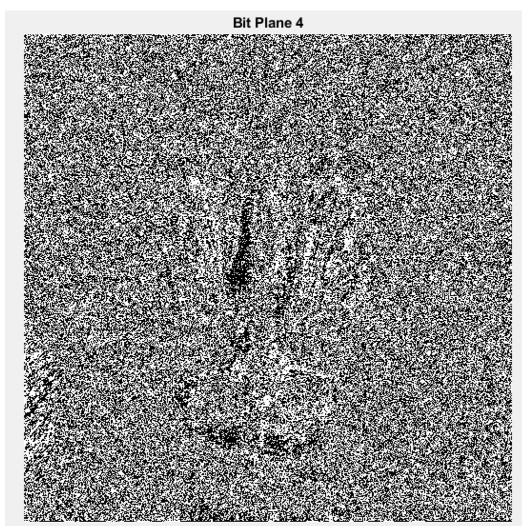

Figure 2: Bit plane 4 of the original baboon image

LSBwkm1 has hidden information within the picture. It contains the image of the Drexel Logo within it’s second bit plane. The second image, LSBwmk2 also contains a hidden content. There is a treasure map hidden within the 1st bit plane of the image. Lastly, there is hidden information with LSBwkm3. Like LSBwmk2, there is a hidden content on the first bit plane of this image. This hidden content is a picture of an alien. These can be seen in figures 3, 4, 5, respectively.

Figure 3: Hidden content found on bit plane 2 of LSBwmk1

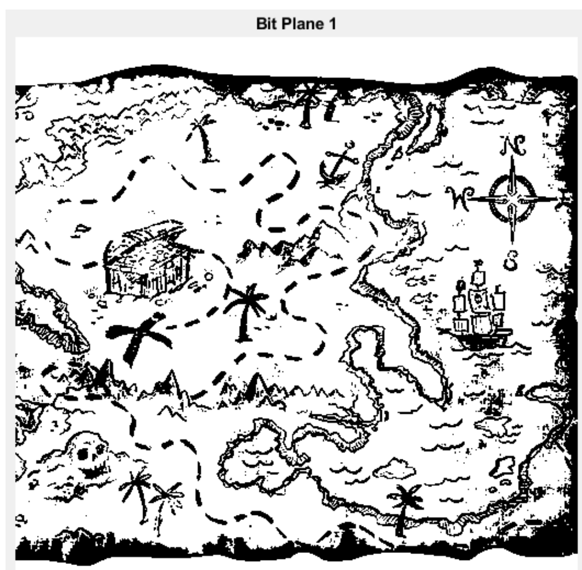

Figure 4: Hidden content found on bit plane 1 of LSBwmk2

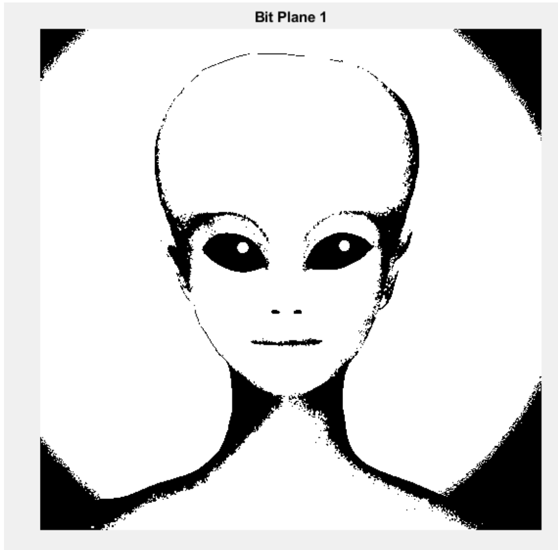

Figure 5: Hidden content found on bit plane 1 of LSBwmk3

For both images, distortion starts to appear at around 4 bit planes of Barbara placed in. However, each image can hide a different amount of bit planes from Barbara before the hidden content starts leaking through the image. For peppers it is 5 bit planes and for baboon it is 6 bit planes. The reason for this is the amount of information needed by the baboon picture to appear properly. The baboon picture relies on it's higher bit planes to show the proper image. This means more of that information can be replaced before it starts to be seen. For peppers, it needs to use some of it's lower bit planes to appear properly. Therefore the hidden content of Barbara starts to appear sooner in the peppers image, because that necessary information is in a lower bit plane compared to baboon. Both images with the hidden content can be seen in figures 6 and 7. Figure 6 shows the peppers image with Barbara hidden in it. Figure 7 shows the baboon image with Barbara hidden in it.

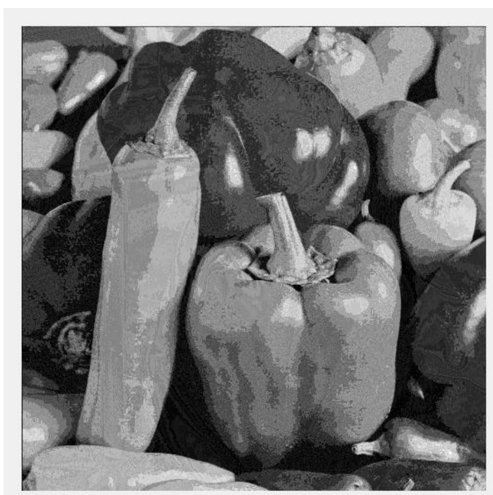

Figure 6: Image of peppers with the 5 LSB planes replaced by the 5 MSB planes of Barbra

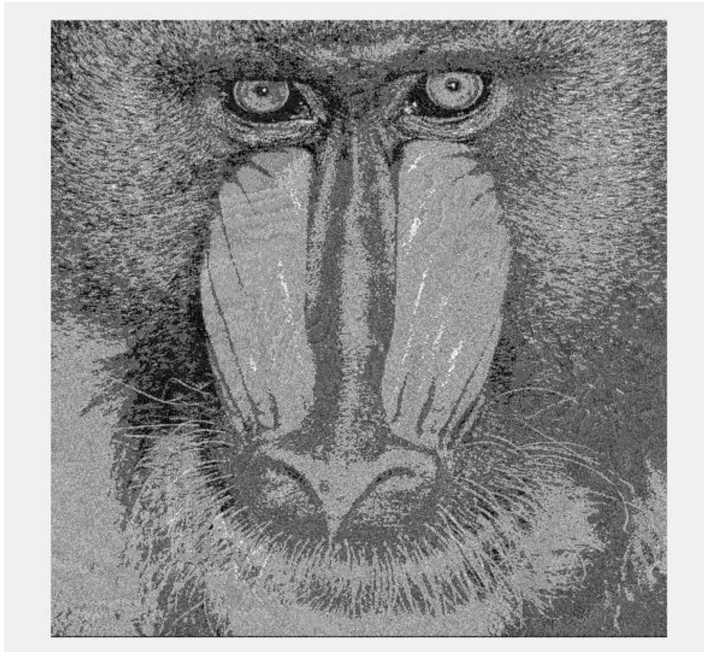

Figure 7: Image of Baboon with the 6 LSB planes replaced by the 6 MSB planes of Barbra

Part 2 – Yeung-Mintzer Watermarking

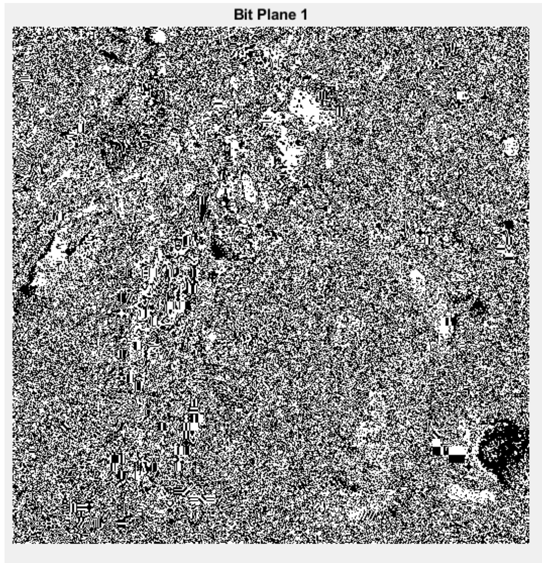

Figure 8: LSB plane of peppers image encoded with water mark

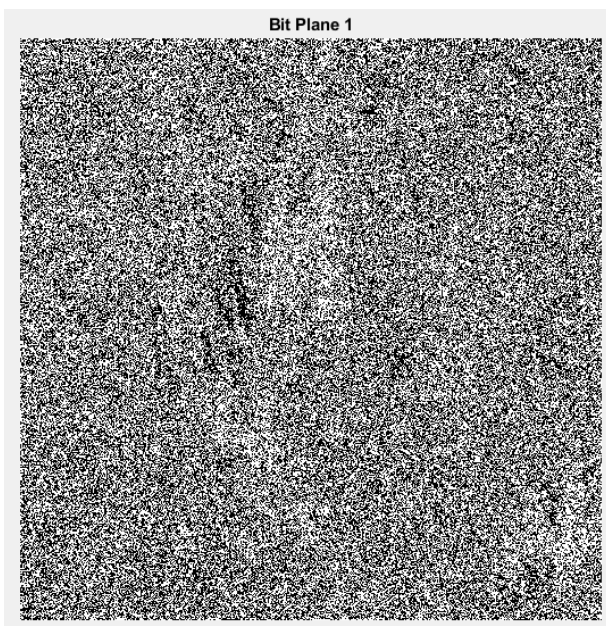

Figure 9: LSB plane of baboon image with encoded water mark

In both the peppers image and the baboon image, the watermark is not detectable using the first code. By looking at the figures above, the LSB plane for both images look similar to that of their non encoded original counterparts. While these are only 2 pictures, it is noticeable that this method does not detect the water mark.

$PSNR_{peppers}$ (original image vs. YM watermarked image): 47.5118

$PSNR_{baboon}$ (original image vs. YM watermarked image): 48.5972

$PSNR_{peppers}$ (original image vs. LSB watermarked image): 51.1422

$PSNR_{baboon}$ (original image vs. LSB watermarked image): 51.1391

The PSNR for the LSB watermarked image will be higher due to the fact that the watermark is more diffused throughout the image. The watermarking function for the Yeung-Mintzer algorithm does not necessarily change every pixel, but instead only changes the pixel values that do not correspond to the watermark. The LSB watermarking method, however, changes the entire bitplane to be the watermarked image. As such, it would be apparent that the LSB watermarking method would introduce greater distortion into the image.

Figure 10: Watermark extracted from peppers_encoded.tiff

Figure 11: Watermark extracted from baboon_encoded.tiff

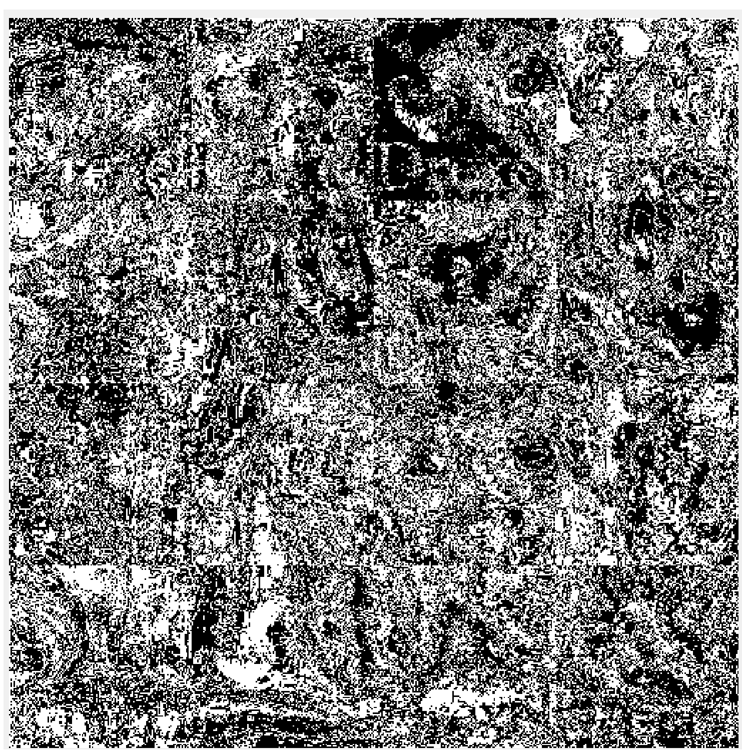

Figure 12: Watermark extracted from YMwmkedKey435.tiff

Figure 13: Combined peppers_encoded.tif and baboon_encoded.tif

Figure 14: Watermark extracted from combined peppers and baboon

Given that the look-up table is identical for both images (peppers and baboon), the extracted watermark should be a combination of both watermarks (as shown above).

PART 1 - Least Significant Bit Data Hiding

```
%The purpose of this code is to hide the N most significant bits of one
%image with the N least significant bits of another image
function replace = hidden(image1, image2, N)

%This function takes an input of 2 images and N, where image 1 is the image
%who's information will be replaced, and N is the number of bit planes
%transfered

image1 = double(image1); %makes the first image a double
image2 = double(image2); %makes the second image a double

p1 = [];
p2 = [];

[r1,c1]=size(image1);

for Z= 1:8
  for Q=1:r1
      for W=1:c1
          p1(Q,W,Z)=bitget(image1(Q,W),Z); %gets the bit plane for the first image,
where each Z value for p1 represents a 512 x 512 bit plane
      end
  end
end

[r2 c2] = size(image2);

for T= 1:8
  for Y=1:r2
      for P=1:c2
          p2(Y,P,T)=bitget(image2(Y,P),T); %gets the bit plane for the second image,
where each T value for p2 represents a 512 x 512 bit plane
      end
  end
end

for B = N:-1:1
    math = 8+1-B; %place holder to match the bit planes correctly
    p1(:,:,B) = p2(:,:,math); %replaces the N least sigficant bit planes of image 1
with the N most significant bit planes of image 2
end

calc = 0;
for ii = 1:8
    calc = calc+ p1(:,:,ii)*(2^(ii-1)); %puts the image back together
end

replace = uint8(calc); %returns image in proper type
end
```

PART 2 - Yeung-Mintzer Watermarking

```
function encoded_image = ym_watermark(image, binary_watermark, key)

    % image = image to place the watermark into
    % binary_watermark = most significant bit plane of watermark image that you
    %                    wish to embed into <image>
    % key = seed the random number generator with the user specified key

    rng(key)        % seed the random number generator with the user specified key
    lut = rand(1,256) > 0.5;     % generate look-up table values

    [num_rows, num_cols] = size(image);

    encoded_image = image;  % make sure encoded_image and image are the same size and
type
    for i = 1:num_rows      % for each row
        for j = 1:num_cols  % for each pixel of each row
            if (binary_watermark(i,j) == 0)
                % unchanged, embed same pixel value
                encoded_image(i,j) = image(i,j);
            elseif (binary_watermark(i,j) == 1)
                % changed, embed closest lut value
                encoded_image(i,j) = lut_lookup(key,
binary_watermark(i,j),double(image(i,j)));
            end
        end
    end
end


function index = lut_lookup(key, value_you_need, n)

    % key = seed the random number generator with the user specified key
    % value_you_need = either a 1 or 0, depending on what the current watermark
    %                  pixel value is
    % n = the index you're trying to find the closest value_you_need to

    % this function returns the index of the closest 1 or 0 to the current index
    % you're looking at

    rng(key)        % seed the random number generator with the user specified key
    lut = rand(1,256) > 0.5;     % generate look-up table values
    idx_needed = find(lut == value_you_need);    % all the places where the lut is
equal to the value you need

    positive_index = min(idx_needed((idx_needed - n) > 0));      % closest positive
index

    negative_index = max(idx_needed((idx_needed - n) < 0));      % closest negative
index

    % contains some error catching for beginning and end of lut
```

```
    if (n == 1) || isempty(negative_index)
        index = positive_index;
    elseif (n == length(lut))
        index = negative_index;
    elseif ((positive_index - n) < (n - negative_index))
        % if the distance between the current index you're looking at and the
        % positive index is less than the distance between the current index and the
        % negative index, set the index to the positive index
        index = positive_index;
    else
        index = negative_index;
    end
end

function watermark = ym_decode(watermarked_image, key)

    % watermarked_image = image that contains the watermark you wish to extract
    % key = seed the random number generator with the user specified key

    % watermark = watermark that has been extracted from watermarked_image


    rng(key)        % seed the random number generator with the user specified key
    lut = rand(1,256) > 0.5;     % generate look-up table values

    [num_rows, num_cols] = size(watermarked_image);     % dimensions of
watermarked_image

    watermark = watermarked_image;  % make sure watermark and watermarked image are
the same size and type

    for i = 1:num_rows      % for each row
        for j = 1:num_cols  % for each pixel of each row
            if watermarked_image(i,j) == 0
                % if the pixel value is equal to 0, set the watermark value to 0
                watermark(i,j) = 0;
            elseif lut(watermarked_image(i,j)) == 0
                % if the lut at index (i,j) is 0, set the watermark index to 0
                watermark(i,j) = 0;
            elseif lut(watermarked_image(i,j)) == 1
                % if the lut at index (i,j) is 1, set the watermark index to 1
                watermark(i,j) = 1;
            end
        end
    end
end
```