Leila Suasnabar
3109-9816-56
lsuasnab@usc.edu

EE 569: Homework #2

Problem 1 : Geometric Image Modification

1.1 Motivation

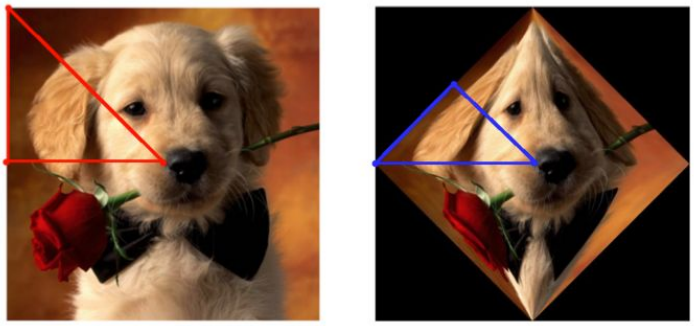
Geometric image modification is one of the most common image processing operations. These operations allow us to scale, warp, rotate, spatially translate and add/or change different perspectives [1]. In this problem, we are asked to perform spatial warping; which can be computed through multiple methods; puzzle matching; which again can be computed by translating, rotating an image or performing affine transformation (triangle mapping); and homographic transformation which adds depth to the image being overlaid on.

1.2 Approach

1.2.1 Geometrical Warping

Two methods were applied to this section. The first one is triangle mapping (affine warping), which works by dividing the image in triangles and using a total of 6 control points (3 vertices points from input and 3 vertices points from output) to find the variable mapping matrix which was then applied to all the pixels within that triangle input region. Both images were segmented into eight triangles.

Visual reference in another image:



H matrix on each triangle (ai,bi are computed using the 3 input vertices and 3 output vertices):

a1	a2	a3
b1	b2	b3
0	0	1

Linear warping equation: [Xi,Yi,1]' = H*[Ui,Vj,1]'.
(Ui and Vj represent the vertices location on the desired output. And Xi and Yi represent the vertices location on the input.)
Set up on Matlab:

```
iPoints(:,:,1) = [249,249,1;249,0,1;0,0,1]';
iPoints(:,:,2) = [249,249,1;0,0,1;0,249,1]';
iPoints(:,:,3) = [249,249,1;0,249,1;0,499,1]';
iPoints(:,:,4) = [249,249,1;0,499,1;249,499,1]';
iPoints(:,:,5) = [249,249,1;249,499,1;499,499,1]';
iPoints(:,:,6) = [249,249,1;499,499,1;499,249,1]';
iPoints(:,:,7) = [249,249,1;499,249,1;499,0,1]';
iPoints(:,:,8) = [249,249,1;499,0,1;249,0,1]';

oPoints(:,:,1) = [249,249,1;249,0,1;124,124,1]';
oPoints(:,:,2) = [249,249,1;124,124,1;0,249,1]';
oPoints(:,:,3) = [249,249,1;0,249,1;124,374,1]';
oPoints(:,:,4) = [249,249,1;124,374,1;249,499,1]';
oPoints(:,:,5) = [249,249,1;249,499,1;374,374,1]';
oPoints(:,:,6) = [249,249,1;374,374,1;499,249,1]';
oPoints(:,:,7) = [249,249,1;499,249,1;374,124,1]';
oPoints(:,:,8) = [249,249,1;374,124,1;249,0,1]';


for i=1:8
    H(:,:,i) = iPoints(:,:,i) * (oPoints(:,:,i)^-1);
    H(:,:,i) = H(:,:,i)^-1;
end
```

With outputs (H for each triangle in ascending order):

```
val(:,:,1) =

    0.5020    0.0000  124.0000
   -0.4980    1.0000  124.0000
   -0.0000    0.0000    1.0000

val(:,:,2) =

    1.0000   -0.4980  124.0000
   -0.0000    0.5020  124.0000
    0.0000   -0.0000    1.0000

val(:,:,3) =

    1.0000    0.4960 -123.5040
   -0.0000    0.5000  124.5000
         0         0    1.0000

val(:,:,4) =

    0.5020   -0.0000  124.0000
    0.5020    1.0000 -125.0000
    0.0000   -0.0000    1.0000

val(:,:,5) =

    0.5000   -0.0000  124.5000
   -0.5000    1.0000  124.5000
    0.0000   -0.0000    1.0000

val(:,:,6) =

    1.0000   -0.5000  124.5000
   -0.0000    0.5000  124.5000
         0         0    1.0000

val(:,:,7) =

    1.0000    0.5020 -125.0000
   -0.0000    0.5020  124.0000
   -0.0000   -0.0000    1.0000

val(:,:,8) =

    0.5000    0.0000  124.5000
    0.4960    1.0000 -123.5040
    0.0000    0.0000    1.0000
```

The second method used a more direct approach. Every row from the top to the middle was obtained proportionally from the original image. Through this the first row was minimized to a pixel, the second row to two pixels, until the 248th row, the next two rows were kept the same and then the row position was inversely proportional to the width of the final result.

Row Number	Column positions used
1	250
2	166,366
3	125,250,375
…	...
250	All column positions
...	...
3	125,250,375
2	166,366
1	250

1.2.2 Puzzle Matching

In order to do the puzzle matching, we first need to find the coordinates of the corners of the pieces, to do this all points were checked against a predetermined set of matrices that represent a possible corner point. Since some of the borders pixels blend in with the background, this was also taken into account and a threshold of 250 was set determine a pixel was valuable. Similarly, a more simple process was done to find the corner points from the images that need to be filled. The larger piece from pieces.raw was separated into two triangles to perform affine warping (triangle mapping) onto the Hillary.raw image. For the smaller piece, bilinear interpolation was first applied to generate pixels at fractional positions when upscaling it. From this point, the steps used on Hillary.raw were repeated on Trump.raw. In order to obtain a smooth result, the pieces were transformed into a slightly bigger area needed, hence the output vertices were modified accordingly. This allowed the less significant pixels from pieces.raw to not be taken into account. This image superposition allowed to obtain a smooth result in the overall filled image. The superposed pixels were ignored if they were of low values compared to the pixels of the original images. Finally, a weighted mean filter was applied on the borders to get rid of unwanted pixels and to smooth the edges. In this problem, the size of the holes in the images was assumed to be known as 100x100, therefore only the top left vertix was computed algorithmically.

Bilinear Interpolation equation:

$$P(p',q') = (1-b)*[(1-a)*P(p,q)+a*P(p+1,q)] + b[(1-a)*P(p,q+1)+a*P(p+1,q+1)]$$

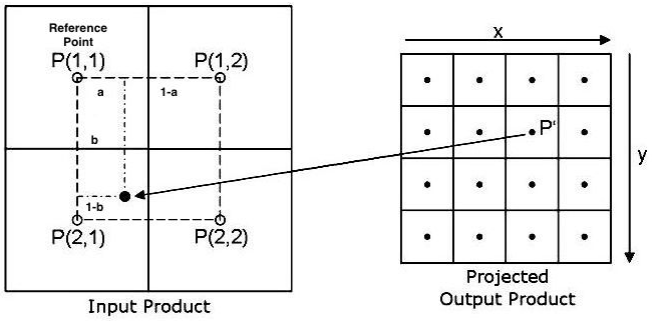
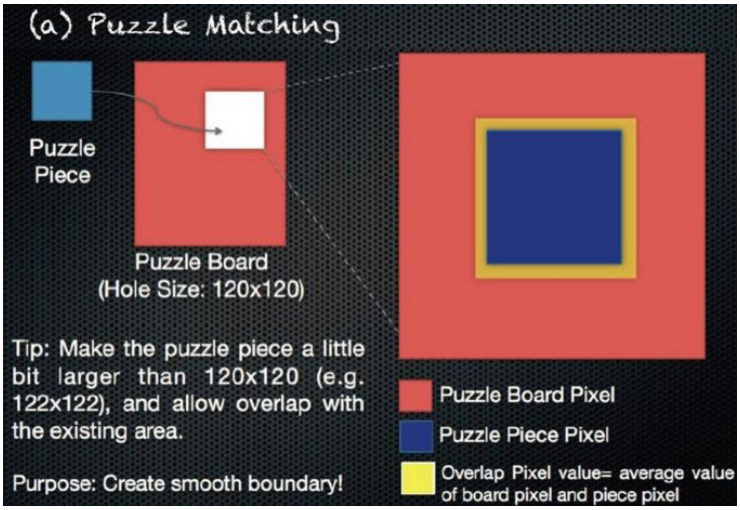


Fig 1. Geometric representation of the bilinear interpolation [2]

Following the algorithm for affine warping (triangle mapping), we use the vertices to find the warping matrix.

```
PUZZLE MATCHING LOG

First image: Hillary.raw
Second image: Trump.raw

Corner pixels of first hole:
Top left: 135, 173
Top right: 135, 272
Bottom left: 234, 173
Bottom right: 234, 272

Corner pixels of second hole:
Top left: 236, 163
Top right: 236, 262
Bottom left: 335, 163
Bottom right: 335, 262

Corner pixels of first piece:
Top left: 57, 96
Top right: 95, 240
Bottom left: 202, 57
Bottom right: 240, 202

Corner pixels of second piece:
Top right: 377, 290
Top left: 302, 296
Bottom right: 383, 364
Bottom left: 308, 371

Size of isolated second piece: 85
New resized image size: 120

Corner pixels of second piece after being resized:
Top right: 2, 108
Top left: 11, 2
Bottom right: 107, 117
Bottom left: 117, 11
```

Diagram of superposition of images [3]:



Affine warping matrices for the first piece (left) and the second piece (right)

```
val(:,:,1) =

    0.6439   -0.1699  114.6083
    0.1744    0.6484  100.8106
    0.0000    0.0000    1.0000

val(:,:,2) =

    0.6453   -0.1691  114.2826
    0.1756    0.6436  100.8462
   -0.0000    0.0000    1.0000
```

```
val(:,:,1) =

   -0.0709    0.9577  232.7940
   -0.9570   -0.0802  274.7306
   -0.0000    0.0000    1.0000

val(:,:,2) =

   -0.0722    0.9563  232.9535
   -0.9650   -0.0893  275.7582
   -0.0000   -0.0000    1.0000
```

1.2.3 Homographic Transformation and Image Overlay

The homographic transformation procedure is stated below. Images of points in a plane, from two different camera viewpoints, under perspective projection (pin hole camera models) are related by a homography:

$$P_2 = HP_1$$

where H is a 3x3 homographic transformation matrix, P_1 and P_2 denote the corresponding image points in homogeneous coordinates before and after the transform, respectively. Specifically, we have

$$\begin{bmatrix} x_2' \\ y_2' \\ w_2' \end{bmatrix} = \begin{bmatrix} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \\ H_{31} & H_{32} & H_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \quad and \quad \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} \frac{x_2'}{w_2'} \\ \frac{y_2'}{w_2'} \end{bmatrix}$$

We assume we know the positions of the 4 vertices from the output image (X2,Y2), and that H33 =1. First the input image is upscaled with bilinear interpolation, this is to avoid empty pixels at

possible fractional points. Upon, obtaining the new resized image positions, we set up the eight linear equations to obtain the homographic transformation matrix. Below is the setup used in Matlab:

```
syms H11 H12 H13 H21 H22 H23 H31 H32
%H11*x1+H12*y1+H13-H31*x1*x2-H32*y1*x2 == x2
%H21*x1+H22*y1+H23-H31*x1*y2-H32*y1*y2 == y2

eqn1 = H11*0+H12*0+H13-H31*0*595-H32*0*595 == 595;
eqn2 = H21*0+H22*0+H23-H31*0*322-H32*0*322 == 322;
eqn3 = H11*0+H12*524+H13-H31*0*428-H32*524*428 == 428;
eqn4 = H21*0+H22*524+H23-H31*0*541-H32*524*541 == 541;
eqn5 = H11*294+H12*524+H13-H31*294*431-H32*524*431 == 431;
eqn6 = H21*294+H22*524+H23-H31*294*650-H32*524*650 == 650;
eqn7 = H11*294+H12*0+H13-H31*294*610-H32*0*610 == 610;
eqn8 = H21*294+H22*0+H23-H31*294*525-H32*0*525 == 525;


[A,B] = equationsToMatrix([eqn1,eqn2,eqn3,eqn4,eqn5,eqn6,...
    eqn7,eqn8], [H11,H12,H13,H21,H22,H23,H31,H32]);

H = linsolve(A,B);
```

And its results:

```
A =

[   0,   0, 1,   0,   0, 0,       0,       0]
[   0,   0, 0,   0,   0, 1,       0,       0]
[   0, 524, 1,   0,   0, 0,       0, -224272]
[   0,   0, 0,   0, 524, 1,       0, -283484]
[ 294, 524, 1,   0,   0, 0, -126714, -225844]
[   0,   0, 0, 294, 524, 1, -191100, -340600]
[ 294,   0, 1,   0,   0, 0, -179340,       0]
[   0,   0, 0, 294,   0, 1, -154350,       0]

B =

 595
 322
 428
 541
 431
 650
```

```
H =

      -895/15974
  2261283/5210132
             595
     17848/29829
  1783675/1302533
             322
    -171/974414
    9163/5210132
```

The resulting matrix H is set as:

-0.056029	0.434016	595
0.598344	1.369395	322
-0.000175	0.001756	1

Finally we need to perform, overlay over the output image as the base and the input image as the top layer image.Overlay algorithm [4]:

$$f(a,b) = \begin{cases} 2ab, & \text{if } a < 0.5 \\ 1-2(1-a)(1-b), & \text{otherwise} \end{cases}$$

where *a* is the base layer value and *b* is the top layer value.

1.3 Results

1.3.1 Geometrical Warping

Fig 1. Left: Original images. Center: Warping using Triangle Mapping. Right: Warping using proportional row warping method.

1.3.2 Puzzle Matching

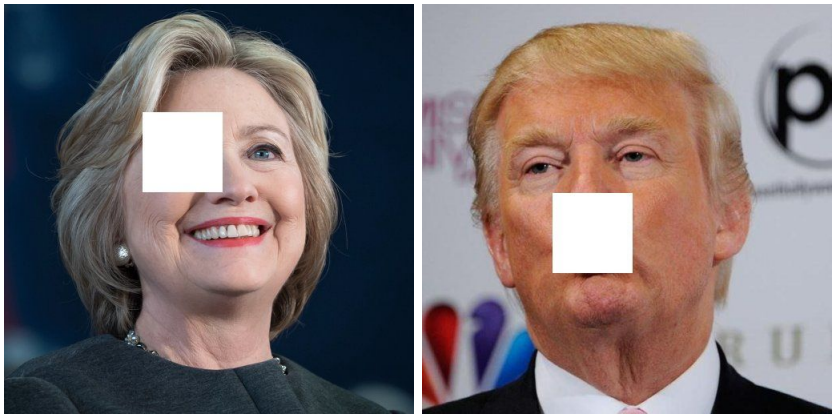


Fig 2. Top: Original images. Bottom: Images after the puzzle matching algorithm

1.3.3 Homographic Transformation and Image Overlay

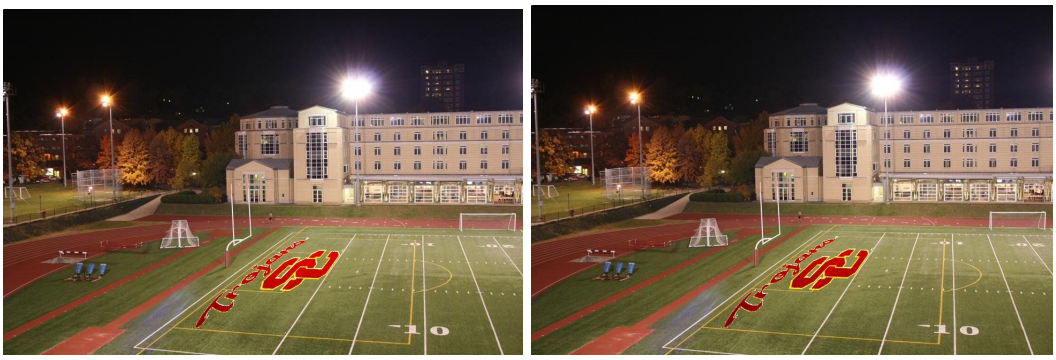


Fig 3. Left: Image after being transformed and overlaid. Right: Output with a slight change in the overlay algorithm

1.4 Discussion

1.4.1 Geometrical Warping

When using triangle mapping, the output looks distorted and although it was warped in a diamond shape it did not look as the expected output. A different method was applied, on which every row was minimized to fit into the diamond shape, this proved a far better option compared to the results of the affine transformation since its output was smoother and less distorted. See fig 1.

1.4.2 Puzzle Matching

As can be seen on fig 2, the algorithm implemented successfully filled the holes with the pieces. The piece from Hillary.raw in Pieces.raw was large enough to fit in the hole after doing affine transformation. However, the piece from Trump.raw needed to be upscaled using bilinear interpolation from 85x85 to 120x120 in order to fill out the fractional positions that the non scaled image showed when performing the affine transformation directly. Since both images

were spatially translated to fit a slightly bigger hole, this allowed to use the most valuable pixels only. As can be seen, the smoothing filter allowed to have an output that has an unchanged contrast among the pixels. Notice, that the pixels at the hole in the Trump.raw solved puzzle looks a little pixelated, this is because the image was upscaled and thus the piece image look a little less sharp.

1.4.3 Homographic Transformation and Image Overlay

The image overlap technique used on this problem and homographic transformation allowed to put an embedded text image (trojans.raw) on a host image (field.raw). The resulting image text desired region was highly sensitive to the accuracy of the homographic transformation matrix, to avoid any problems the matrix's elements was set to at least six decimal places. A threshold of 250 was set to ignore the background, and the overlay algorithm was added to obtain the output displayed in Fig 3 Left. An experiment was employed to make the overlay more visible, the overlaying image was minimized by 5% (Fig 3. Right), although this allowed the output image to show a more notorious overlaying, as expected the image being overlaid looked a little less color saturated; nonetheless, both images still show the overlaying process and the homographic transformation on the desired region.

Problem 2 : Digital Halftoning

2.1 Motivation

Dithering allows to add more dimension to an image by creating the illusion of using grayscale color. This technique allows to convert a grayscale image to a binary image, which can be useful when outputting images with binary values. Therefore, image rendering wise, the dot density in the image can be changed to appear grayscale. Also this technique can be implemented to avoid adding noise by adjusting the threshold values in the algorithm.

Another good method for digital halftoning is error diffusion. This method pushes the noise/error push to unprocessed nearby pixels only. By doing so, the algorithm reduces the quantity of quantization levels [5]. Digital halftoning 'makes the image suitable for printing on binary printers such as black and white laser printers"[5].

2.2 Approach

2.2.1 Dithering Matrix

For this problem, the image was first expanded by copying the edge rows and columns to obtain enough pixels to be able to pass the dithering filter on every pixel from the original image. Then the dithering matrix was computed with the following:

The Bayer index matrices are defined recursively using the formula:

$$I_{2n}(i,j) = \begin{bmatrix} 4 * I_n(x,y) & 4 * I_n(x,y) + 2 \\ 4 * I_n(x,y) + 3 & 4 * I_n(x,y) + 1 \end{bmatrix}$$

The index matrix can then be transformed into a threshold matrix T for an input gray-level image with normalized pixel values (*i.e.* with its dynamic range between 0 and 255) by the following formula:

$$T(x,y) = \frac{I(x,y) + 0.5}{N^2} \times 255$$

$$G(i,j) = \begin{cases} 1 & \text{if } F(i,j) > T(i \bmod N, j \bmod N) \\ 0 & otherwise \end{cases}$$

where *F(I,j)* and *G(I, j)* are the normalized input and output images.
There are different ways to construct a dithering matrix. Another example for a 4x4 matrix is:

$$A_4(i,j) = \begin{bmatrix} 14 & 10 & 11 & 15 \\ 9 & 3 & 0 & 4 \\ 8 & 2 & 1 & 5 \\ 13 & 7 & 6 & 12 \end{bmatrix}$$

For this problem 4 kernels are used, A (shown above), and I with sizes 2x2, 4x4, and 8x8. Only the I2x2 and A4x4 were hard coded, the others matrices were computed with the code. From the image above, G was renormalized back to the 0-255 range in order to show the image.

Kernels used:

0	2
3	1

0	8	2	10
12	4	14	6
3	11	1	9
15	7	13	5

0	32	8	40	2	34	10	42
48	16	56	24	50	18	58	26
12	44	4	36	14	46	6	38
60	28	52	20	62	30	54	64
3	35	11	43	1	33	9	41
51	19	59	27	49	17	57	25
15	47	7	39	13	43	5	37
63	31	55	23	61	29	53	21

For the final section, four gray level colors were used and the following statements were used to display the new output:

```
if (imageData[i][j]>5*T/3)
    imageOut[i][j] = 255;

else if(imageData[i][j]>4*T/3 && imageData[i][j]<=5*T/3)
    imageOut[i][j] = 170;

else if(imageData[i][j] > T && imageData[i][j] <= 4*T/3)
    imageOut[i][j] = 85;
else
    imageOut[i][j] = 0;
```

Where imageData represents the array of the input image “man.raw”, imageOut is the array containing the output image, and T is the threshold.

2.2.2 Error Diffusion

Similarly to the previous part, the image was first expanded as explained in part 2.2.1. For this part, the 4 type of filters were applied to every pixel of man.raw (while centered in these kernels). In order to preserve the quantization error, the rows are serpentine scanned.

- Floyd-Steinberg’s matrix:

$$\frac{1}{16}\begin{bmatrix}0 & 0 & 0\\ 0 & 0 & 7\\ 3 & 5 & 1\end{bmatrix} \qquad \frac{1}{16}\begin{bmatrix}0 & 0 & 0\\ 7 & 0 & 0\\ 1 & 5 & 3\end{bmatrix}$$

Left to Right Matrix Right to Left Matrix

- Jarvis, Judice, and Ninke (JJN)'s matrix:

$$\frac{1}{48}\begin{bmatrix}0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 7 & 5\\ 3 & 5 & 7 & 5 & 3\\ 1 & 3 & 5 & 3 & 1\end{bmatrix}$$

Left to Right Matrix

$$\frac{1}{48}\begin{bmatrix}0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\\ 5 & 7 & 0 & 0 & 0\\ 3 & 5 & 7 & 5 & 3\\ 1 & 3 & 5 & 3 & 1\end{bmatrix}$$

Right to Left Matrix

- Stucki's matrix:

$$\frac{1}{42}\begin{bmatrix}0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 8 & 4\\ 2 & 4 & 8 & 4 & 2\\ 1 & 2 & 4 & 2 & 1\end{bmatrix}$$

Left to Right Matrix

$$\frac{1}{42}\begin{bmatrix}0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\\ 4 & 8 & 0 & 0 & 0\\ 2 & 4 & 8 & 4 & 2\\ 1 & 2 & 4 & 2 & 1\end{bmatrix}$$

Right to Left Matrix

- Proposed method:

$$\frac{1}{16}\begin{bmatrix}0 & 0 & 0\\ 0 & 0 & 1\\ 3 & 5 & 7\end{bmatrix}$$

Left to Right Matrix

$$\frac{1}{16}\begin{bmatrix}0 & 0 & 0\\ 1 & 0 & 0\\ 7 & 5 & 3\end{bmatrix}$$

Right to Left Matrix

2.3 Results

2.3.1 Dithering Matrix

Fig 4. Left: Original man.raw. Center: Dithered with I2x2. Right: Dithered with I8x8

Fig 5. Left: Dithered using A4x4. Right: Dithered using I4x4

Fig 6. Left: Dithered by A4x4 and I4x4. Right: Dithered by I4x4 and A4x4

Fig 7. Left:Dithered by A4x4 using 4 grayscale colors . Right: Dithered by I4x4 using 4 grayscale colors.

2.3.2 Error Diffusion

Fig 8. Top Left: Original man.raw. Top Right: Result of using Floyd-Steinberg's error diffusion. Bottom Left: Result of using JJN's error diffusion. Bottom Center: Result of using Stucki's error diffusion. Bottom Right: Result of new proposed error diffusion method.

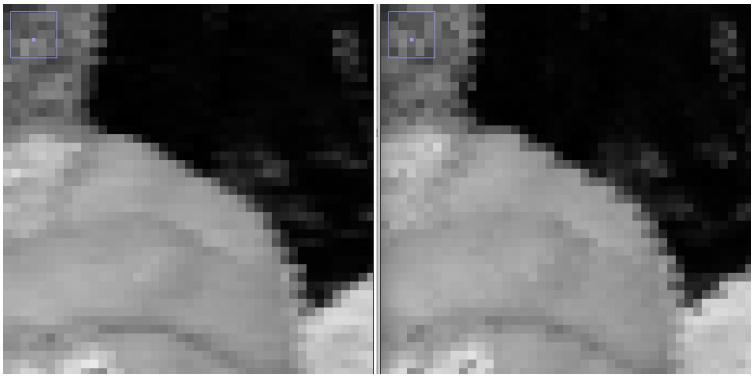


Fig 9. Zoomed in area (man's knuckles). Left: New proposed error diffusion method. Right: Using Floyd-Steinberg's error diffusion method

2.4 Discussion

2.4.1 Dithering Matrix

For the first part, the bayer matrices of size 2x2 and 8x8 are applied to the image (Fig 4). From these results, we see that the result looks better when using the size 2x2 bayer matrix. The 8x8 matrix has fewer valuable pixels compared to the 2x2 one.

For the second part, we apply the bayer 4x4 and A 4x4 dithering matrix on the man.raw. The result of using the bayer matrix stays closer to the contrast of the original image, this can be seen in Fig 5. Although the A 4x4 dithering matrix lighten the image, we can see an overall smoother change between the pixel values. We then apply both filters on the image, this can be seen in Fig 6. From these images, we see that there isn't too much of a significant change between the images after being dithered with the other matrix compared to being dithered just one time.

Lastly, from Fig 7 we see that adding an extra 2 grayscale colors allowed the obtain an output closer to the original image. However, the difference compared to using 2 colors was only slightly better. For this image particularly, the threshold values were computed by giving a larger range to the dark pixels and less emphasis on the 2 middle colors.

2.4.2 Error Diffusion

From Fig 8, we see that the resulting images are significantly better than the ones obtained through dithering. By pushing back the quantization error, we are able to see an output closer to the original image. Notice; however, that the Floyd-Steinberg method shows a more sharp result compared to using JJN and Stucki's method. From the result from using the JJN and Stucki matrix, we see that these two outputs are relatively similar. Because of this results, the matrix for the new proposed method was based on the best resulting method among the three, the Floyd-Steinberg method. For the new method, the matrix used was essentially similar to the Floyd-Steinberg one (see section 2.2.2) with the idea of distributing the most error to the last pixel in the filter instead of the next pixel to be processed. By doing so recursively, we keep

passing most of the error to the last pixel and thus suppress the low frequency noise slightly better. As Fig 9 shows, in this zoomed in comparison between the Floyd-Steinberg and the new method, the new method shows sharper edges.

Problem 3 : Morphological Processing

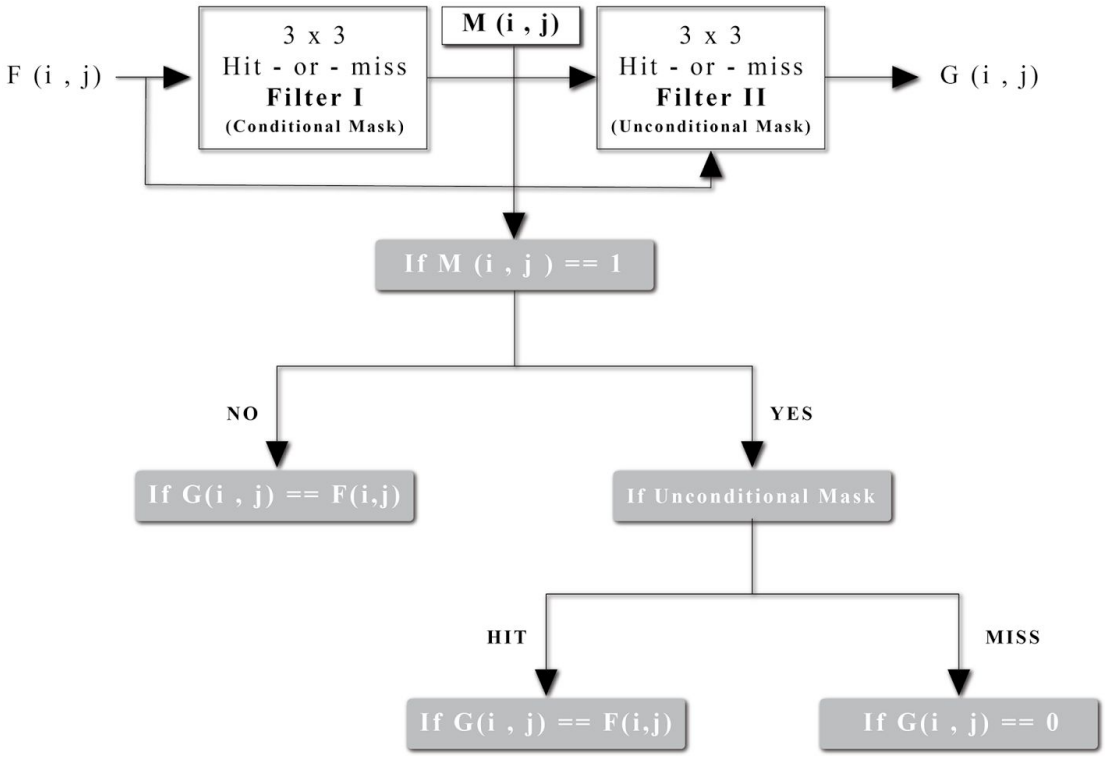
3.1 Motivation

Morphological image processing focuses on binary images. These operations allows us to obtain further data from a picture, such as finding the number of objects in a binary image, check the object's sizes, analyze its shape, etc. For instance, we can obtain information from a handwritten letter or a black and white document. In this problem, we'll see how shrinking, thinning and skeletonizing can be applied to binary images. Additionally, with these operations we can fix images by filling up holes, removing protuberance pixels and remove isolated noise pixels [3].

3.2 **Approach**

3.2.1 Shrinking

All morphological processing operations used in problem 3, were implemented as a two stage system and applied to binary images. Before going into any stage, the image is first expanded by adding an extra row and column on all four sides. The first stage runs every neighborhood of 3x3 pixels (as a bit string representation) if the center pixel is ON or has a value of 255. This bit string is then compared to preset filters [conditional masks] that will determine the pixels that are good candidates to be removed. These new pixels are stored as a new matrix, and then while comparing to every ON pixel from the original image, the new pixels are stored as a bit string and compared to advanced filters [unconditional masks] to determine if they can be turned OFF (set to 0). Upon acquiring the new shrinked image, this procedure is repeated until the image converges. Finally, the image was resized back to its original size.



Flowchart of the two phase system

Upon finding the shrinked image, we ran a 3x3 filter to find single pixels, if there are any matches then we can count that pixel as a square.

0	0	0
0	1	0
0	0	0

Mask to find single pixels

3.2.2 Thinning

The same shrinking procedure is applied to the image, except we use different sets of stage I and stage II filters in order to make objects thinner instead of converging them to a single point.

3.2.3 Skeletonizing

The same shrinking procedure is applied to the image, except we use different sets of stage I and stage II filters in order to find the skeleton of the images instead of converging them to a single point.

3.2.4 Counting Game

First, we expand the image with black pixels in order to be able to use the morphological filters on every pixel. To find the number of holes (black circular holes within white objects), we invert the binary image and apply the shrinking morphological operation. Upon obtaining the new image, we then ran the same algorithm to count single pixels, this will give us the amount of holes within the picture. Additionally, we store the locations of all these single points which we now are pixels that belong to the black holes.

Now, we use these pixels location as starting points to fill out the black holes within the image board.raw. To do this, we use the following information:

$$X_k = (X_{k-1} \oplus B) \cap A^c$$

$$B = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}, X_0 = p$$

Where p is the initial interior point, these points were found in the previous part, and A is the input image. We repeat this procedure until we reach convergence:

$$X_k = X_{k-1}$$

And finally perform this last step:

$$X_{conv} \cup A$$

Lastly, to determine the shape of the object, we find the area and perimeter of the objects in order to get the circularity which will determine if an object is a square, circle or other.
We first isolate every object, we do so by taking the center pixel of every object; which can be found by shrinking the hole filled image to a point; and expanding a square around it until it's edges are all black pixels. For this method to work, it was assumed that the objects were not that close to each other. Once every object was boxed in, we proceed to count the number of nonzero pixels for this will determine its area. To find the perimeter, we looked at every 4-connectivity neighborhood of every nonzero pixel, and counted the number of zero pixels (edge sides).

Implemented procedure

1	0	0
1	1	1
1	1	1

More accurate procedure

1	0	0
1	1	1
1	1	1

However, this method is only accurate with 4-connectivity edge pixels (as seen above), since the implemented algorithm won't take into account diagonal sides. Therefore this method is only used to know if an object is squared or not, since the perimeter of a square object was assumed to not depend on diagonal sides for this problem. Due to this, a separate algorithm was implemented to analyze each object and determine if it's a circle or not. To do this, the computed area was divided by the expected area: $\Pi * r^2$, with r as the radius of the object. A threshold was set as: 0.85 < ratio < 1.05 to determine it's a circle.

3.3 **Results**

3.3.1 Shrinking

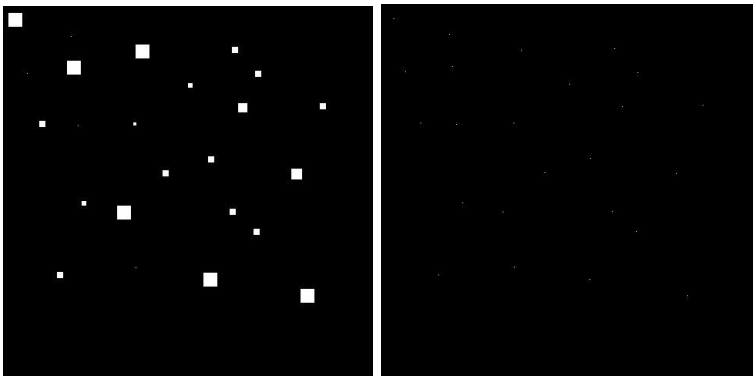


Fig 10. Left: Original image. Right: Shrinked image

```
SQUARES LOG

Total number of squares: 24

Number of iterations: 0, Number of squares: 4, with a Size of: 1x1
Number of iterations: 1, Number of squares: 0, with a Size of: 2x2
Number of iterations: 2, Number of squares: 1, with a Size of: 4x4
Number of iterations: 3, Number of squares: 2, with a Size of: 6x6
Number of iterations: 4, Number of squares: 9, with a Size of: 8x8
Number of iterations: 5, Number of squares: 0, with a Size of: 10x10
Number of iterations: 6, Number of squares: 1, with a Size of: 12x12
Number of iterations: 7, Number of squares: 1, with a Size of: 14x14
Number of iterations: 8, Number of squares: 0, with a Size of: 16x16
Number of iterations: 9, Number of squares: 6, with a Size of: 18x18
```

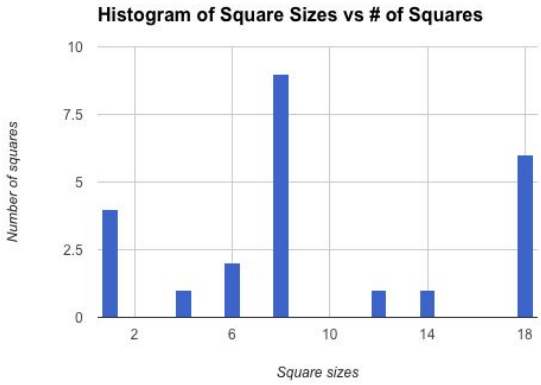


Fig 11. Left: Description of the squares. Right: Histogram of square size vs frequency

3.3.2 Thinning

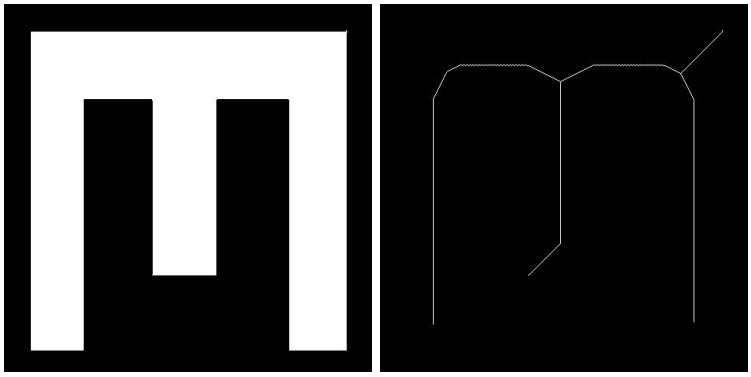


Fig 12. Left: Original image. Right: Thinned image

3.3.3 Skeletonizing

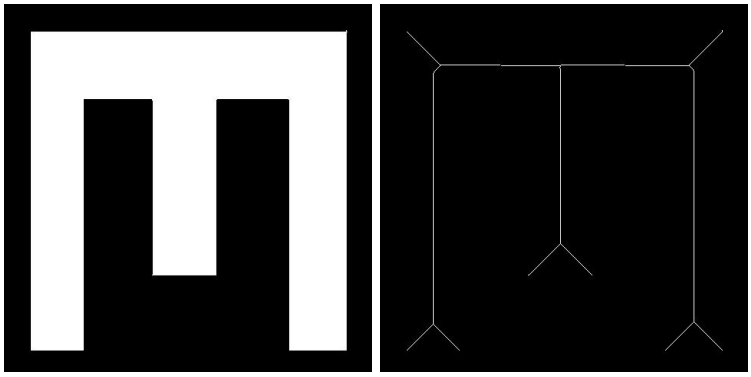


Fig 13. Left: Original image. Right: Skeletonized image

3.3.4 Counting Game

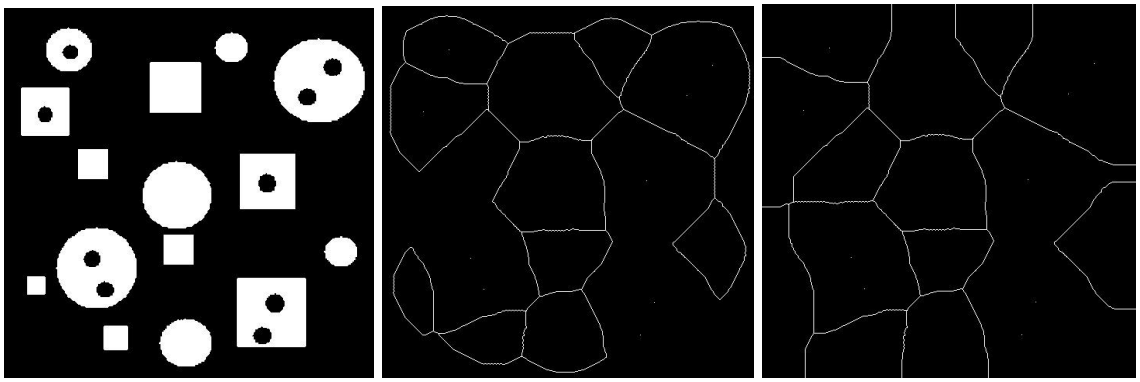


Fig 14.Left: Original board.raw. Center: Inverted and then expanded with black pixels, and finally shrinked. Right: Expanded with black pixels and then inverted, and finally shrinked

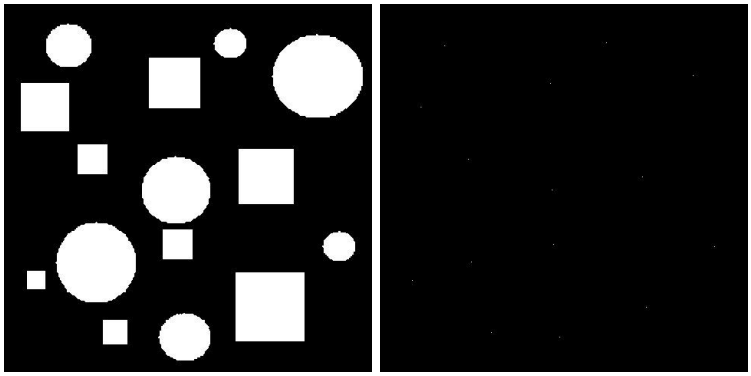


Fig 15. Left: Image with filled holes. Right: Left image shrinked

```
COUNTING GAME

There are 9 holes within white objects in the image

There are 15 white objects in the image

Object 1 has an area of 1281, and a perimeter of 164
And a circularity of 0.59851
Object's shape is a circle

Object 2 has an area of 2686, and a perimeter of 232
And a circularity of 0.627104
Object's shape is a circle

Object 3 has an area of 10064, and a perimeter of 458
And a circularity of 0.602906
Object's shape is a circle

Object 4 has an area of 4420, and a perimeter of 266
And a circularity of 0.784999
Object's shape is a square

Object 5 has an area of 3965, and a perimeter of 252
And a circularity of 0.784607
Object's shape is a square

Object 6 has an area of 1521, and a perimeter of 156
And a circularity of 0.785398
Object's shape is a square

Object 7 has an area of 5184, and a perimeter of 288
And a circularity of 0.785398
Object's shape is a square

Object 8 has an area of 6176, and a perimeter of 358
And a circularity of 0.605552
Object's shape is a circle

Object 9 has an area of 1521, and a perimeter of 156
And a circularity of 0.785398
Object's shape is a square

Object 10 has an area of 1281, and a perimeter of 164
And a circularity of 0.59851
Object's shape is a circle

Object 11 has an area of 8557, and a perimeter of 424
And a circularity of 0.598136
Object's shape is a circle

Object 12 has an area of 574, and a perimeter of 96
And a circularity of 0.782671
Object's shape is a square

Object 13 has an area of 8096, and a perimeter of 360
And a circularity of 0.78501
Object's shape is a square

Object 14 has an area of 1020, and a perimeter of 128
And a circularity of 0.78233
Object's shape is a square

Object 15 has an area of 3293, and a perimeter of 266
And a circularity of 0.584842
Object's shape is a circle

-> The total number of circles in the image is: 7
-> The total number of squares in the image is: 8
```

Fig 16. Description of the objects and quantity of circles and squares

3.4 **Discussion**

3.4.1 Shrinking

As shown in Fig 10, the shrinking algorithm shrinked every object to a single pixel. This allowed us to then simply count the single pixels in order to find the total number of objects in the image, which was 24. Furthermore, the sizes were also computed and shown in a histogram (Fig 11). For this part the iterations represent the number of times the algorithm was performed

in order to shrink the image to a single pixel. Also, it was assumed the image had only squares and that these were not connected to one another.

3.4.2 Thinning

Fig 12 shows the image after being thinned using the two phase algorithm. From this output, we can see that in the original image there were some protuberances on the edges that were shown when thinned.

3.4.3 Skeletonizing

Fig 13 shows the image after being thinned using the two phase algorithm. From this output, we can see that skeleton kept the outer shape of the image which allows us to see what the width of the letter was originally.

3.4.4 Counting Game

As Fig 14 shows, upon shrinking the inverted image we counted the single pixels to determine the number of holes: 9 (see Fig 16). The positions of these allowed us to fill the holes and use this new image for the next parts. The new filled image is shrinked so that we can find the number of white objects in the image: 15. The circularity value was only used to find if there were any squares within the image: 8, and the number of circles: 7 was computed using the ratio of area computed/area expected (as explained in 3.2.4). Similarly to the assumption in 3.4.1, it was assumed that the objects were not connected or in close proximity. Additionally, particularly in this image, there was one rectangle but since the size difference between width and length was minimal (one pixel), it was treated as square.

References

[1] G. Wolberg, *Digital Image Warping*, IEEE Computer Society Press, Washington DC, 1990.
[2]Aamna Saeed , "Multi-Resolution Mosaicing", *Computer Science and Engineering*, Vol. 2 No. 6, 2012, pp. 101-111. doi: 10.5923/j.computer.20120206.04.
[3] USC EE569 Spring 2017 Discussion slides and HW descriptions, and Lecture notes.
[4]https://en.wikipedia.org/wiki/Blend_modes
[5]https://en.wikipedia.org/wiki/Error_diffusion