#5	Solutions to Homework 1	172a	Pamela Cosman	1/25/12

1. **Morphological operators on a hexagonal lattice:**

(1) S={1}, L(a,b)=b, n=1, f=1 goes with (M) Reproduces the input image without change (identity operator)
Since L(a,b)=b, it doesn't matter whether surround 1 is found or not. The output is just equal to the input b.

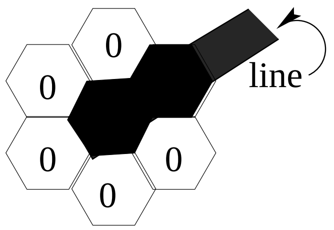
(2) S={}, L(a,b)=ab, n=1, f=1 goes with (I) Resets all cells to zero
Since S is the empty set, a=0 always.

(3) S={1}, L(a,b)=ab, n=1, f=1 goes with (C) Keeps only isolated cells that are one.
The output is equal to 1 only if the input pixel *b* is equal to one AND the surround is all zeros.

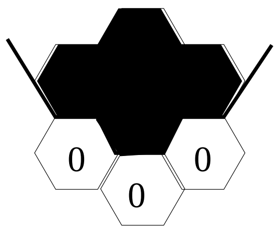
(4) S={2}, L(a,b)=b$\bar{a}$, n=1, f=1 goes with (L) Removes ends of lines
Here the output pixel is equal to zero if the input pixel is a zero. In addition, the output pixel is equal to zero if the input pixel was a 1, and if also exactly one of its neighbors is one. In this case, it can be considered the end of a line, and it is getting removed. See the figure below left.

(5) S={7}, L(a,b)=ab, n=1, f=1 goes with (D): Removes edges of blobs, keeping the interior
If the input pixel is a zero, then it stays a zero. But if it's a one, it gets converted to a zero UNLESS all the neighbors are one. That means a pixel can stay equal to one only if it is in the interior, surrounded on all sides by ones.

(6) S={4}, L(a,b) = ab, n=1, f=1 goes with (P): Marks all corners, flushes the rest
Here the output is equal to one only if the input pixel is one, AND also the surround consists of 3 adjacent ones on one side, and 3 adjacent zeros on the other side. This particular pattern can be considered a corner (see the figure below right), in which case the "corner" pixel gets marked with a 1, and anything else gets zero.

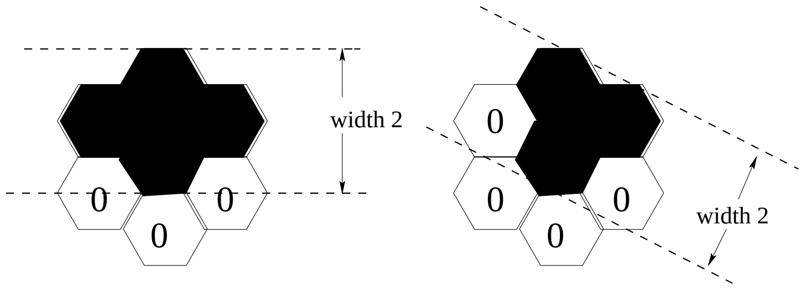


End of a line.



A corner

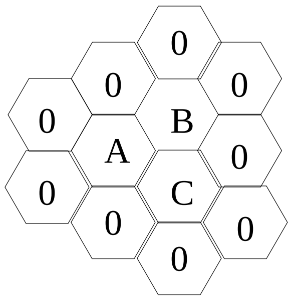
(7) S={3,4}, L(a,b) = $\bar{a}$b, n=∞, f=3 goes with (N): Skeletonizes until lines are only one picture cell wide
The output is a 1 only if the input pixel is a 1 and also the surround is not 3 or 4. Any time the center pixel is equal to one, and the surround is either 3 or 4, we have a situation where the picture item is greater than one cell wide:



By removing that center cell (changing it from a one to a zero) we are removing the double width. Of course, there are other patterns as well that have double width. For example, pattern 5 with the center cell equal to 1. But here we can expect that there would be other cells removed first. The lower left cell of surround 5, for example, may be the center cell of surround 3, and would therefore get removed that way. Subsequently, the remaining pattern would look like a surround 4.

Since we have $n = \infty$, the process is continued until the picture item is down to a one-cell-wide skeleton.

The reason for the 3-fold tesselation is that we do not want to wipe out any patterns entirely. Suppose we have something that looks like this:



where all the pixels A,B,C are equal to 1. If they are all processed simultaneously, then each of the three will get erased, as each has a surround that is a (rotated) version of surround 3. If one of them goes first however, and gets removed, then the others will not get removed, since they no longer find themselves with surround 3.

(8) S={1,2,3}, L(a,b) = $\bar{a}$b, n=∞, f=1 goes with (G): Cuts off all appendages (that is, thin lines) and removes tiny blobs
In this case, the output is 1 only if the input is 1 and also the surround is not one of 1,2,3. Surround 1 means we have an isolated 1-cell; this gets removed. Surround 2 means we have the end of a line; this gets removed. Surround 3 means we have a small appendage protruding from a larger item; this also gets removed. The smallest item which does not get eaten away is a compact blob of 7 one's (central cell and its 6 neighbors). Several people thought this

operation was nearly the same thing as (g): removing edges. But in fact, this operator does not remove (at all) the edges of a large blob.

(9) S={5,6,7}, L(a,b) = a + $\bar{a}$b, n=∞, f=3 goes with (A): Fills in small cavities
Now we have a function L which is going to increase the number of 1-cells. The output will be one any time the input is one OR the surround is one of 5,6,7. But 5,6,7 are all cases where there is an interior hole (zero) or a "bay" of zeros. These holes and cavities get turned into ones, "filled in."

(10) S={8}, L(a,b) = ab, n=1, f=1 goes with (H) Mark all places where 3 thin lines come together
If 3 non-touching single-pixel lines comes together in one pixel, it would look like surround 8 surrounding a 1.

(11) S={1}, L(a,b) = ab, n=1, f=1 followed by S={7}, L(a,b) = a$\bar{b}$, n=1, f=1 goes with (I) Resets all cells to zeros.
The first operation is the same as (3) above. Only isolated cells that are one are kept. The second operation would mark with a 1 any place where there is an isolated zero found surrounded by ones. However, after the first operation, there won't be any occurrences of these to find with the 2nd operation. So, in the second operation, a=0 always, and the output is zero.

(12) S={7}, L(a,b) = a$\bar{b}$, n=1, f=1, followed by S={1}, L(a,b)=ab, n=1, f=1 goes with (Q) Marks any isolated zero cell
The first operation will mark any isolated zero cell. Marking it means putting a 1 there. The second operation, as before, would retain any isolated one cell, but after the first operation is finished, the only isolated ones appearing will be the things that got marked during the first operation, that is, the isolated zero cells.

(13) S={1,7}, L(a,b)=ab, n=1, f=3 goes with (K) Removes edges of blobs, keeping only the interior, but making sure not to completely erase any blob
This is similar to (5) above, in that edges are being erased. But since we also put surround 1 in the set, any isolated pixel will be kept. Since we put in 3-fold tesselation, no blob can get wiped out completely.

(14) S={1}, L(a,b)=ab, n=5, f=1 goes with (C) Keeps only isolated cells that are one.
With n=1, this is the same as (3) above. Isolated ones are kept. After we do it once, there are only isolated ones left in the picture. Doing it 4 more times doesn't change anything.

(15) S={7}, L(a,b)=ab, n=5, f=1 goes with (E) Removes 5 layers of edge pixels from blobs (for example, any blob whose maximum dimension is 10 pixels or less will be erased)
This is like the erosion operator in (5) above, except we are iterating on it 5 times.

2. The random flipping of p% of pixels can be done like this:

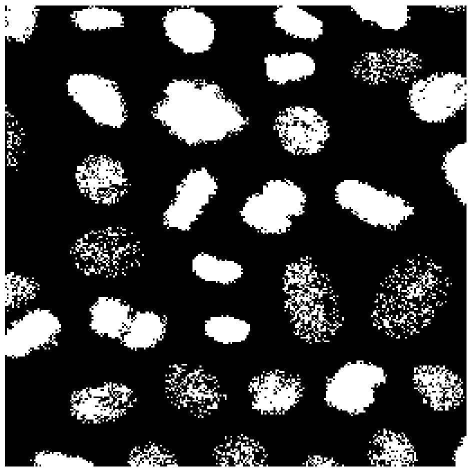
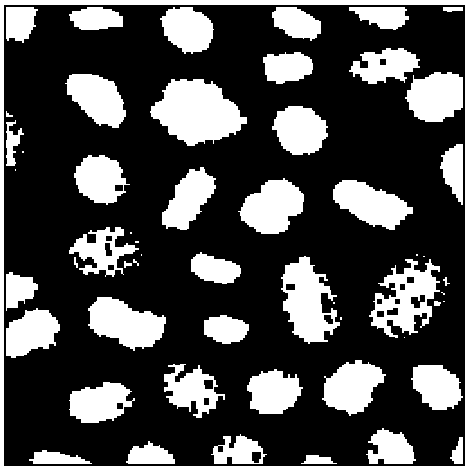
```
function out = ipflip( in, p )
out = xor( in, (rand( size(in) ) < (p/100)) );
```

Isolated pixel removal with the "clean" command works very well. The noise level has decreased significantly while the actual image remained undistroted. However, there is still some noise left. To the human observer, this noise seems to exhibit some granular structure, since the noisy pixels are now grouped into clusters.

Opening followed by closing is effective in removing the noise for $p = 1\%$. In fact all of the noise is gone. For 5% most of the noise is gone, but some pixel groups have survived and have in fact become larger. For 20% noise very many of the pixel groups have grown into larger squares so that the level of noise has actually increased. It is interesting to note, however, that the inside of the letters (or whatever remained of it) is now completely noise free. At the same time the letters suffered distortion. The smaller the features, the more distortion. In fact the smallest letters are unrecognizable or even completely gone.

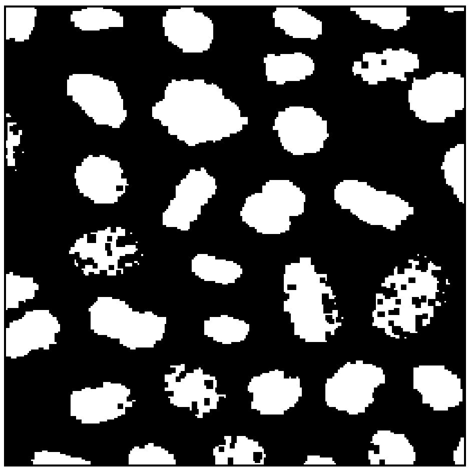
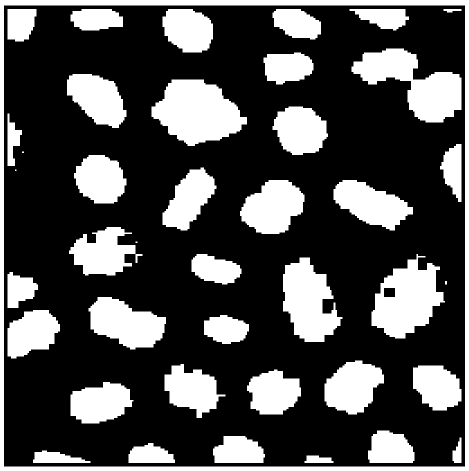
If we do closing first, then opening, the results are slightly different. Now the noise outside of the letters is completely gone. However, so are the smaller letters. In the larger letters, the noise has become worse.

3. First of all, we can try the "close" operator, and that immediately fixes a lot of the problem. Below left is shown the original binary thresholded image, and below right shows the result of using the closing operator on it:

Tiny holes in blobs are filled in, but larger holes are not. If we did two rounds of dilating before two rounds of eroding, some larger holes would fill in, but also a few cells start to merge together. There are several things we can do to prevent this. For example, we can try a spur removal operation, since the cells that merge first appear to do so because of spurs that jut out and then get dilated into causing a merge.

So, I tried the following sequence, starting from the original image: close, spur, dilate, spur, close, erode. The intermediate result after the close and spur operations is shown below left, and the final result after the whole sequence is shown below right:

There are only two cells, near the upper right, which got merged. Of course there are other combinations of operators that you could have used.

4. First I tried looking at the length of the 4-connected perimeter and the 8-connected perimeter. I am using the sum of pixels in the perimeter as being a measure of the length. For the smooth square:

```
>> sum(sum(bwperim(sq,4)))
ans =   160
>> sum(sum(bwperim(sq,8)))
ans =   160
```

I consider defining raggedness by using the ratio of these two numbers.

```
>> sum(sum(bwperim(ragged1,4)))
ans =   156
>> sum(sum(bwperim(ragged1,8)))
ans =   232

>> 156/232
ans =    0.6724

>> sum(sum(bwperim(ragged2,4)))
ans =   220
>> sum(sum(bwperim(ragged2,8)))
ans =   292
```

```
>> 220/292
ans =    0.7534
```

This is not good. It is clearly not monotonic. The ratio for the smooth square is 1, for the slightly ragged square is 0.6724, and for the more ragged square is 0.7534.

I tried a few other things, such as the difference between these two quantities divided by the larger one. Those didn't work either, and so I started thinking more carefully.

An object with a smooth boundary should have the boundary rather unchanged by an opening or closing operation. However, an object with a ragged boundary should have the boundary dramatically changed by an opening or closing operation. This might be the basis of a parameter which measures the boundary raggedness.

```
>> r3 = bwmorph(ragged3,'close');
>> sum(sum(bwperim(r3,8)))
ans =   160
```

I get the same value of 160 for the perimeter of ragged2 and ragged1 after closing. I look at the ratio of the (8-connected) perimeter length after closing to the (8-connected) perimeter length before closing. For the smooth square and the progressively more ragged squares, this ratio is:

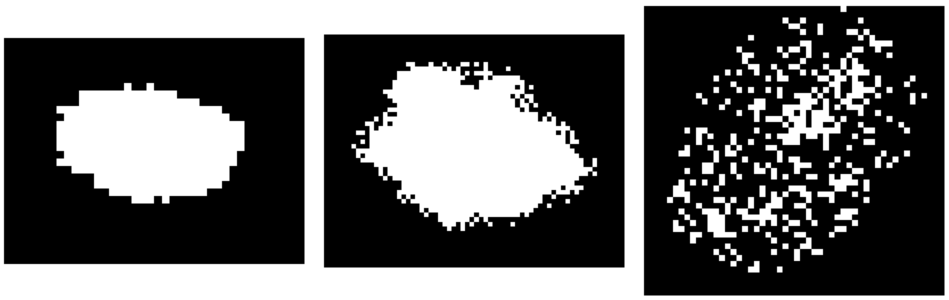
```
    1.0000     0.6897     0.5479     0.4494
```

and we can apply $-log$ to it and produce the result

```
    0.0000     0.3715     0.6017     0.7998
```

Well, this looks good– at least it is monotonic. Many other possible measures of boundary raggedness are also valid, and may be better than this one.

5. Now we try it on 3 cells. The three cells are shown below. On the left is a smooth cell, in the middle is one with a ragged boundary, and on the right is one that is completely fragmented.



Cell 1 has a raggedness value close to 1, as we would like:

```
>> sum(sum(bwperim(cell1,8)))
ans =    81
>> sum(sum(bwperim(bwmorph(cell1,'close'),8)))
ans =    78

>> -log(78/81)
ans =    0.0377
```

For the ragged cell2, we get 0.3911. For the fragmented cell, however, we get a value of -0.1384. So the method breaks down for this case of a disconnected object (not a surprising result).

Some Useful Commands from the TA:

Summation Over Image

Sum up the values of all pixels in image. Some people used two for loops in homework 1 to sum up values in an image. We could use a matlab built-in function **sum** to help us

```
>> sum(sum(Image))
```

The command equals 2-D summation. To make the code more clean, we could first turn the 2-D Image into a 1-D vector and use 1-D summation to sum over it:

```
>> sum(Image(:))
```

The command $Image(:)$ concatenates column vectors in Image to form one single column vector, which has dimension equal to width * height of Image.

Save Figures

The homework requires you to paste in experiment results. It is time consuming to save them manually. We can use the Matlab function **saveas** to save the figure in your scripts.

```
>> h = figure;
>> imshow(Image);
>> saveas(h, 'FILENAME', 'bmp')
```

You can save it as other type such as 'jpeg', 'eps'..., etc.