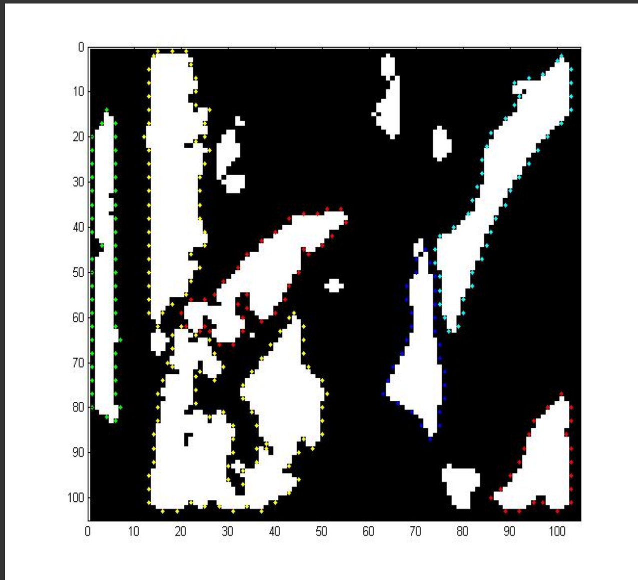
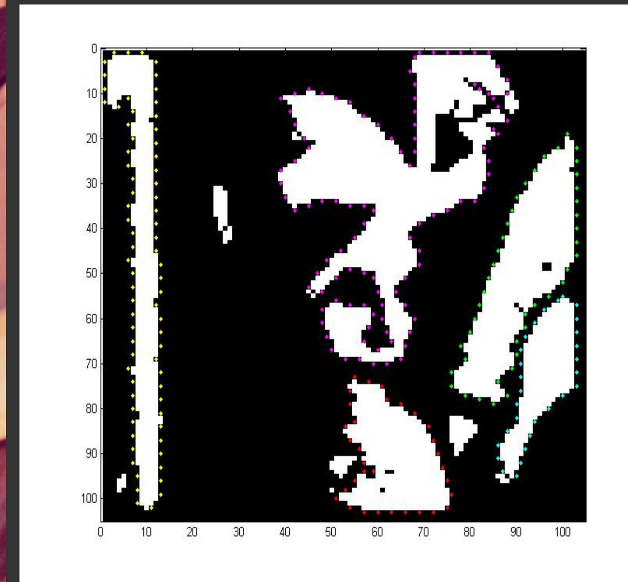



# Boundary Detection in Noisy Images

24 July 2008

Max Mautner
Claremont McKenna College
Claremont, CA

Tarik Trent
Emory University
Atlanta, GA

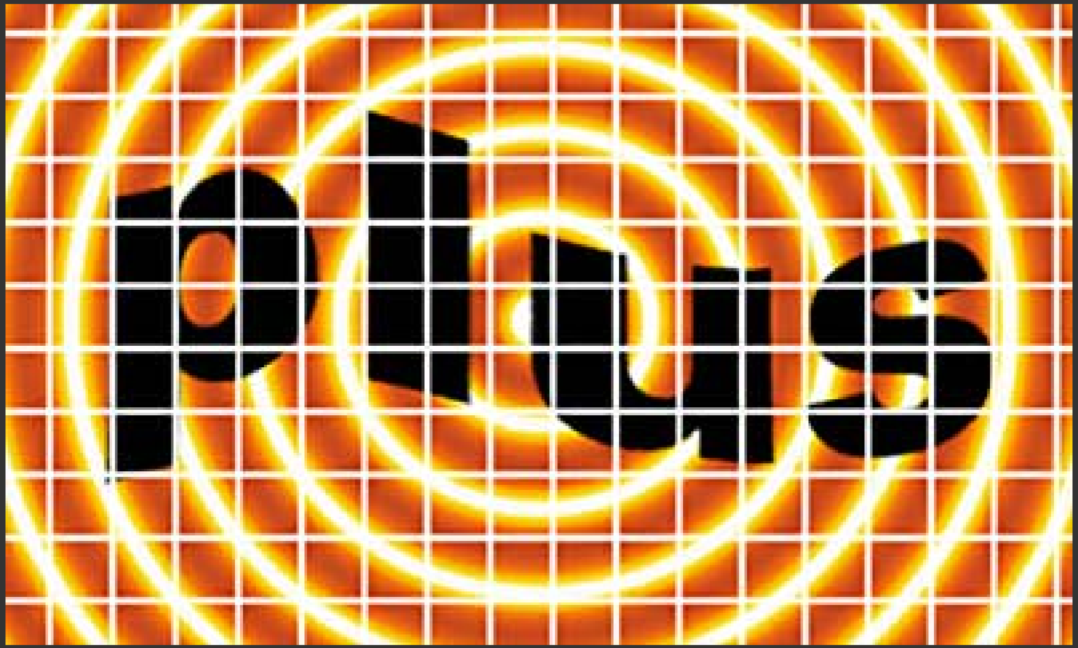
# Intro to Image Processing

**- MathWork's Matlab product:**
**The Image Processing Toolbox**
**- Applications in computer vision, motion analysis, biometrics**
**- Segmentation & edge-finding**


**Dr. Lai's mission for us:**

Find the boundaries of similarly patterned areas in noisy pictures and output them for denoising by spline triangulations.

# Common techniques for segmentation




- Histogram-based
- Clusters
- Region-growing

Gradient-based regions
Contrast peaks and valleys

All such techniques are not directly intended to be used in noised-and-denoised images.

The Berkeley Segmentation Engine

# Whereas we are dealing with this:

# What's the best way to find significant regions in a *noisy* image?

**Image requires additional cleaning**

**Boundary point-collecting**
- Crawling (edge-finding)
- Box-centers (region-growing)
- Iterating over both black and white thresholded images

**Remove redundancy in boundary chains**

**Remove isolated points**

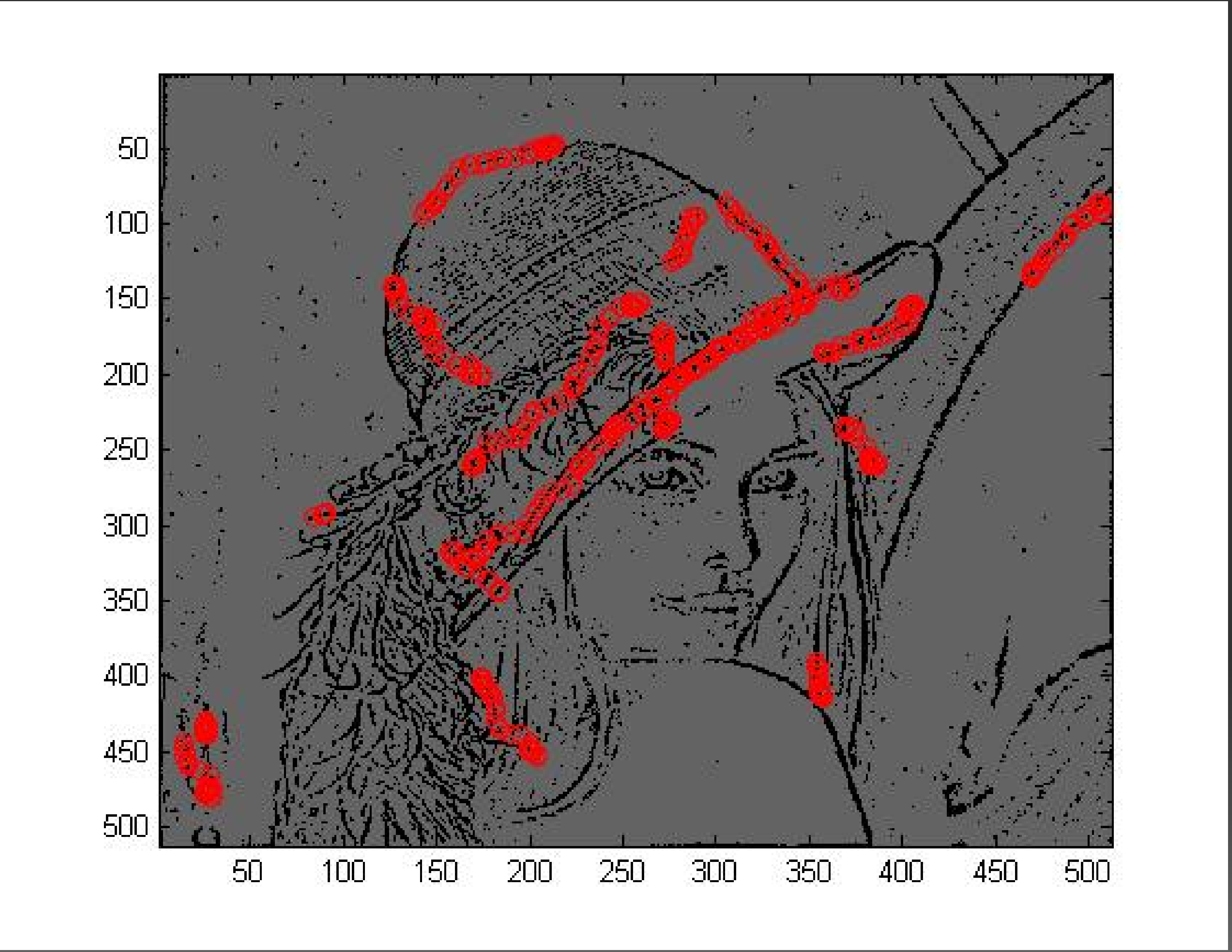
**Order the boundary points**

# First attempt - Edge crawling

```
function [X,Y]=hasNeighbor(A,D,x,y,C1,C2)
    % hasNeighbor takes a binary matrix A and an ordered vector of
    % points around the current point in the crawl.

    S=size(A);
    % default X, Y vals in case no neighbor is found
    X=[x]; Y=[y];
    test=0; % breakout value
    % check for black points in list of D coordinates
    for c=1:length(D)
        %index-out-of-bounds test
        if (D(1,c)<1 || D(1,c)>S(2) || D(2,c)<1 || D(2,c)>S(1))
            ;
        else
            if A(D(2,c),D(1,c))>1
                test=1;
                for i=1:length(C1)  %loop checks for hitting previous point
                    if (C1(i)==D(1,c) && C2(i)==D(2,c))
                        test=0; % prevent breakout, point has already been found
                    end
                end
                if (test==1) % not a repeated point
                    X=[X, D(1,c)]; % store coordinates
                    Y=[Y, D(2,c)];
                end
            end
        end
    end
```

Enmeshedness, recalculating slope, forking directions; it was generally a mess.

# Trial and error - Edge-crawling



# Trial and error - Cleaning the image

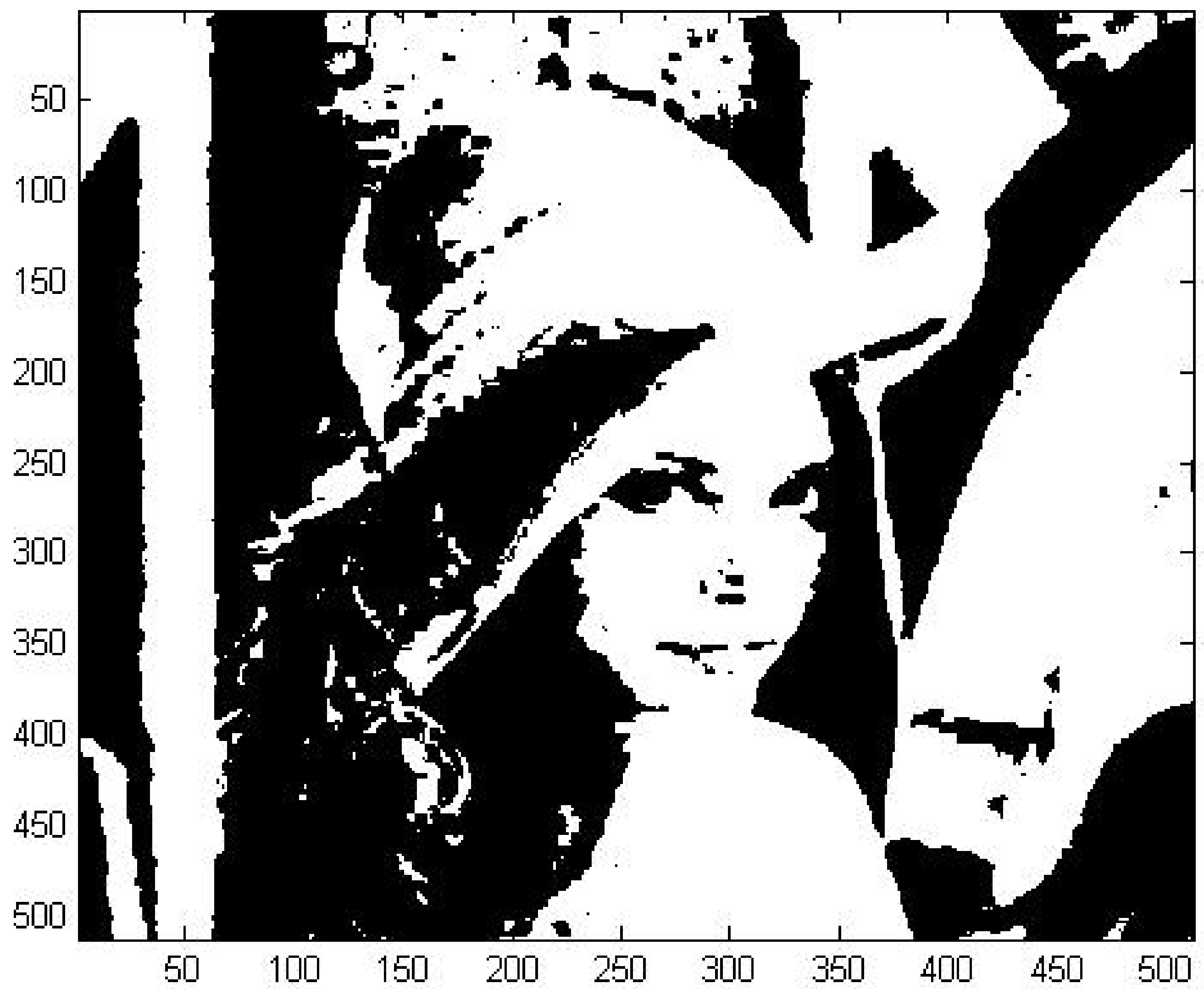
**Thinning**
- How?
- How much?

**Setting thresholds**
- Handpicked
- Automatic
- Both black and white thresholds

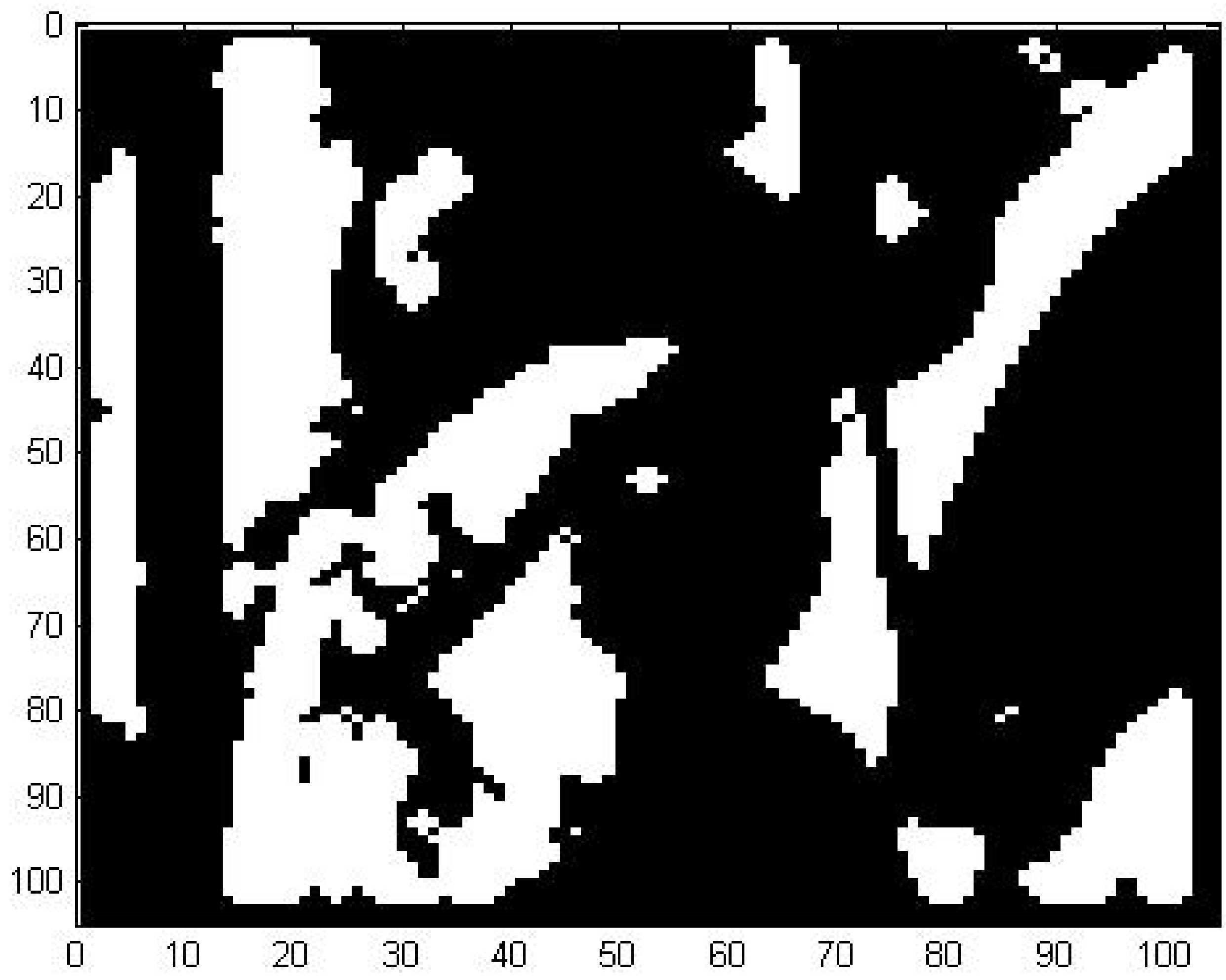
**"Compressing" each 5 x 5 macropixel**
- Identifying solely white areas of the image
- Form regions from these adjacent white areas

Original *Lena* image

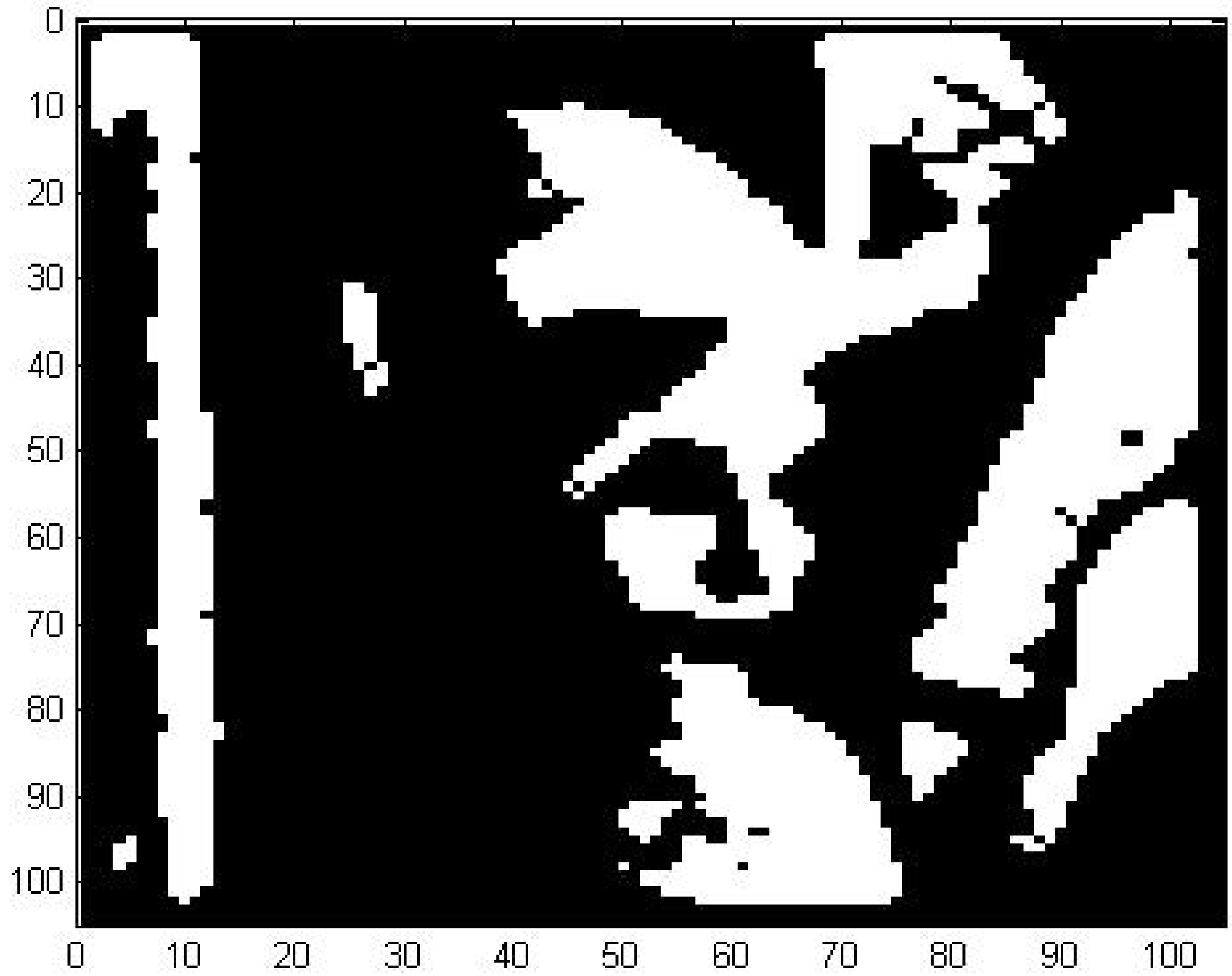

Noised, then denoised *Lena*




*Lena* after thresholding




"Compressed" image of box centers

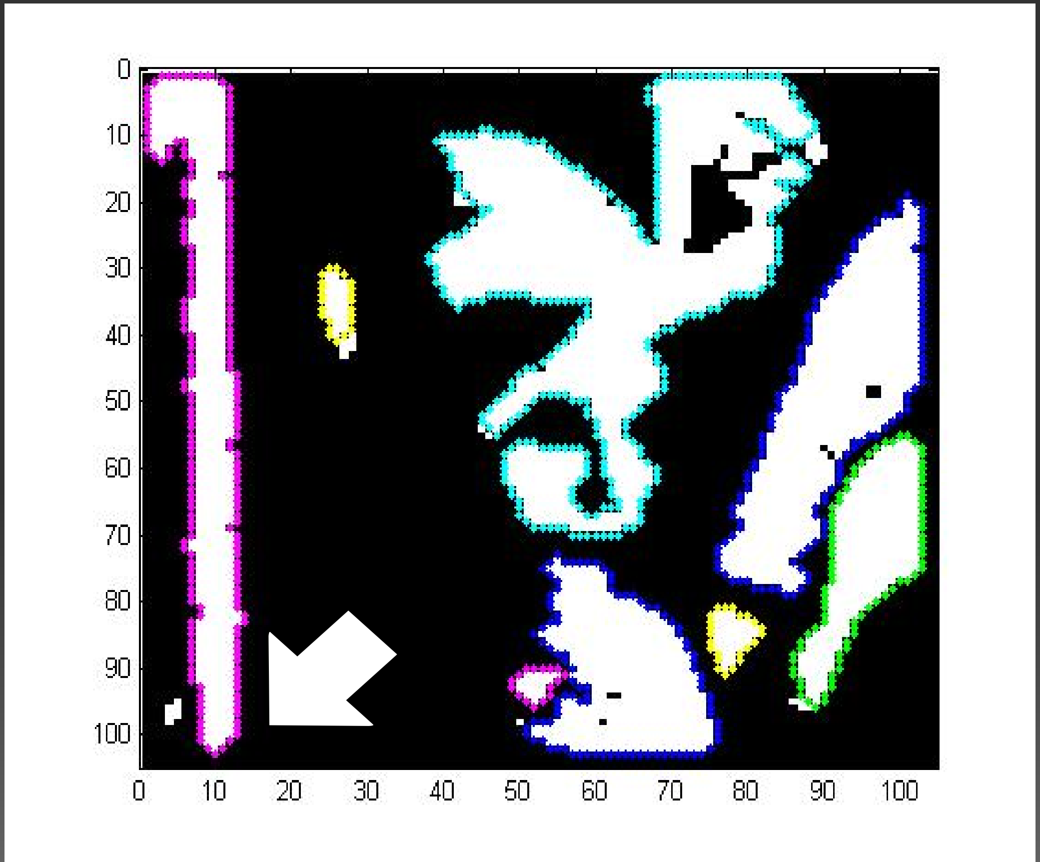


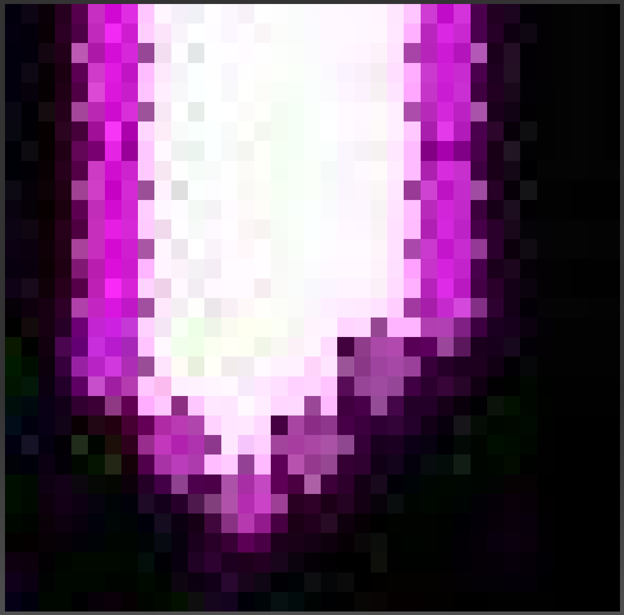

"Compressed" image of box centers (inverted)

# What do we do with these?

Clearly we use these to find the boundaries of the white areas.

We programmed an unseeded region-growing method, where we iterate through every white pixel in the compressed image, initiating a crawl for other white neighbors.



# Removing redundancies

```
function C = cleanChains(C)
% removes redundancy in each chain of cell array C
    Schain=size(C);
    for i=1:Schain(2)
        chain=C{i};
        S=size(chain); count=1;
        while(count<=S(2))
            J=find(chain(1,:)==chain(1,count) & chain(2,:)==chain(2,count));
            j=2;
            while (j <= length(J))
                chain(:,J(j))=[0; 0];
                j=j+1;
            end
            c1=nonzeros(chain(1,:));
            c2=nonzeros(chain(2,:));
            chain=[c1'; c2'];
            count=count+1;
            S=size(chain);
        end
        C{i}=chain;
    end
```

First step of the chain-aggregating process

# Clean isolated points from the chains

```
function E=removeIsolatedPoints(C)
% remove isolated boundary points in each chain of
% the cell array C
    S=size(C);
    E=cell(1,S(2));
    for i=1:S(2)
        chain=C(i);
        SChain=size(chain);
        for j=1:SChain(2)
            if (hasNeighbors(chain,chain(:,j)) <= 1)
                chain(:,j)=[0;0];
            end
        end
        c1=nonzeros(chain(1,:));
        c2=nonzeros(chain(2,:));
        E(i)=[c1'; c2'];
    end
```

If a point has one or no white neighbors, it's irrelevant.

# Ordering of boundary points

```
function finalChain=orderChain2(chain)
% orders the closed boundary coordinates of each cell
% in the cell array 'chain'
    cells=1; SC=size(chain); finalChain={};
    while (cells <= SC(2))
        C=chain{cells};
        orderedChain=C(:,1);
        initPoint=C(:,1);
        point=[-1;-1];
        count=1;
        alreadyChecked=initPoint; % for backtracking
        SChain=size(C);
        while (count < SChain(2))
            if (count==1)
                C(:,1)=[];
                point=initPoint;
            end
            if (count==3)
                C=[initPoint C];
            end
            isntNew=1;
```

```
% down
J=find(C(1,:)==point(1,1) & C(2,:)==point(2,1)-1); %find neighbor
if (isempty(J)==0) % neighbor exists?
   orderedChain=[orderedChain C(:,J)]; % add neighbor to orderedChain
   point=C(:,J); % set new point
   alreadyChecked=[alreadyChecked point]; % add to list of checked
   C(:,J)=[]; % delete found point from unorderedChain C
   isntNew=0; % point's been found, no need to check other neighbors
end

if (isntNew==1)
   %downright
    J=find(C(1,:)==point(1,1)+1 & C(2,:)==point(2,1)-1) ;
    if (isempty(J)==0)
       orderedChain=[orderedChain C(:,J)];
       point=C(:,J);
       alreadyChecked=[alreadyChecked point];
        C(:,J)=[];
       isntNew=0;
    end
end

% etc.
```

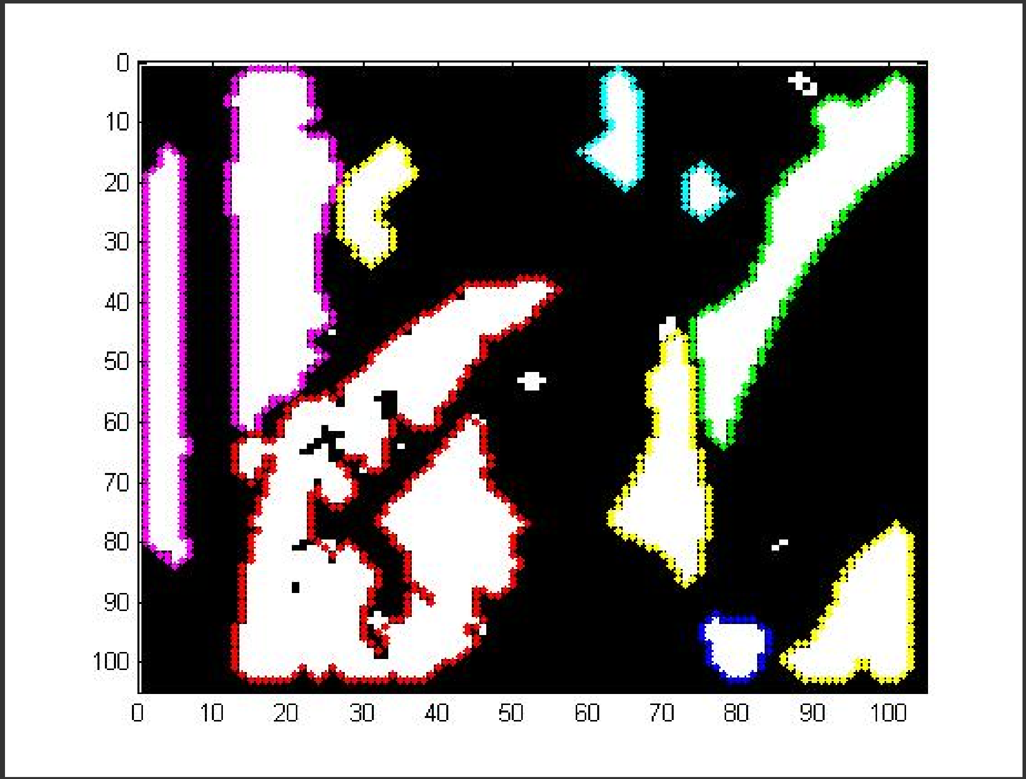
```
            if (initPoint(1,1)==point(1,1) && initPoint(2,1)==point(2,1))
                break;
            end

            % should we go back?
            if (isntNew==1 && isempty(alreadyChecked)==0)
                SChecked=size(alreadyChecked);
                point=alreadyChecked(:,SChecked(2));
                alreadyChecked(:,SChecked(2))=[];
            else
                count=count+1;
            end
        end

        finalChain{cells}=orderedChain; % add chain to cell array
        cells=cells+1; % increment cell array counter
    end

```

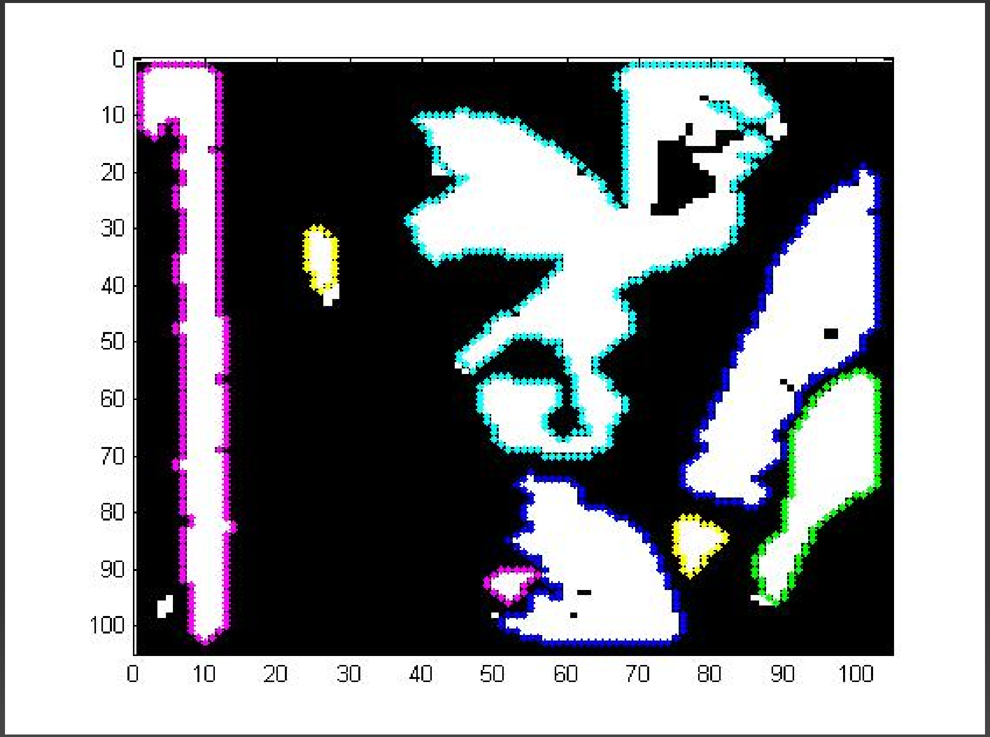
End of the while loop, crawling around boundary

# Thin Chains

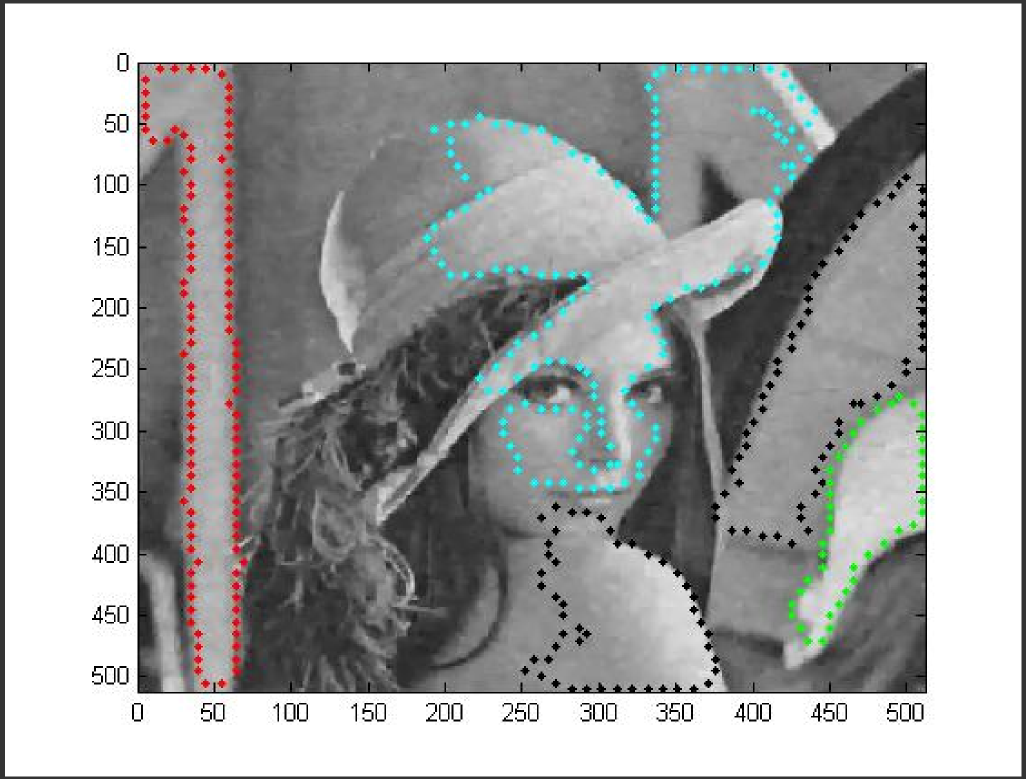
```
function C = thinChains(chain, pointspacing)
 thinchain={};
 sChain=size(chain);
 for i = 1:sChain(2)
        c=chain{i};
        thin=[;];
        for j=1:pointspacing:length(c)
            thin=[thin, c(:,j)];

        end
        thinchain{i}=thin;
 end

 C=thinchain;
```

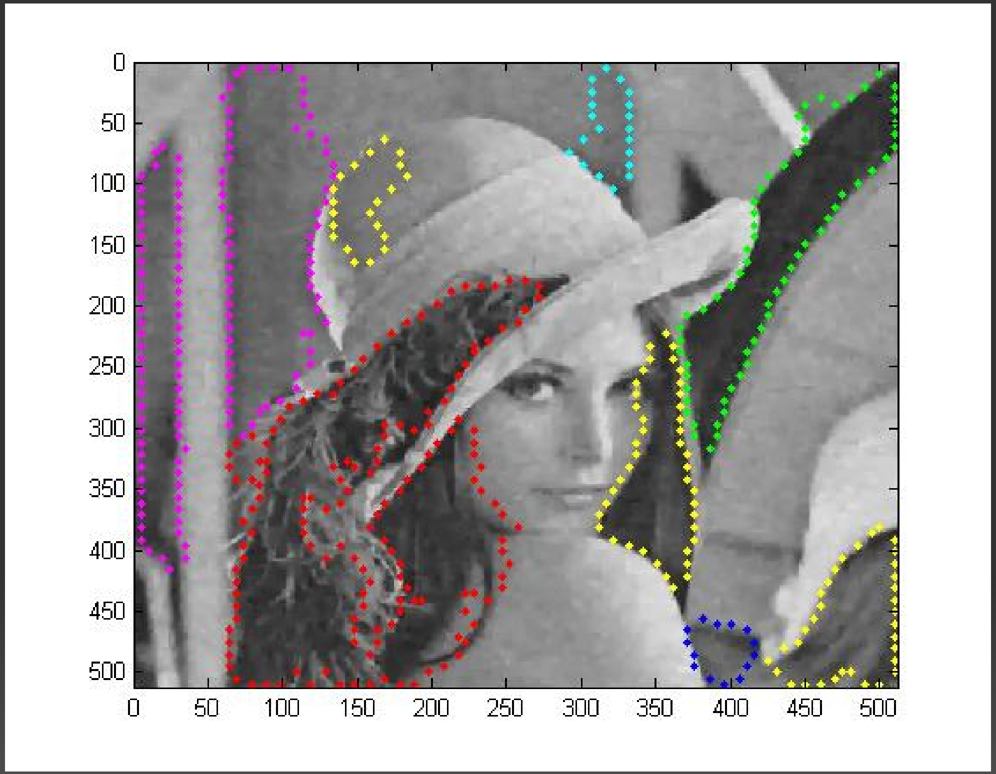
We look at every other point (representing a five-by-five section of the image) in the chain in order to make the triangulation process faster.

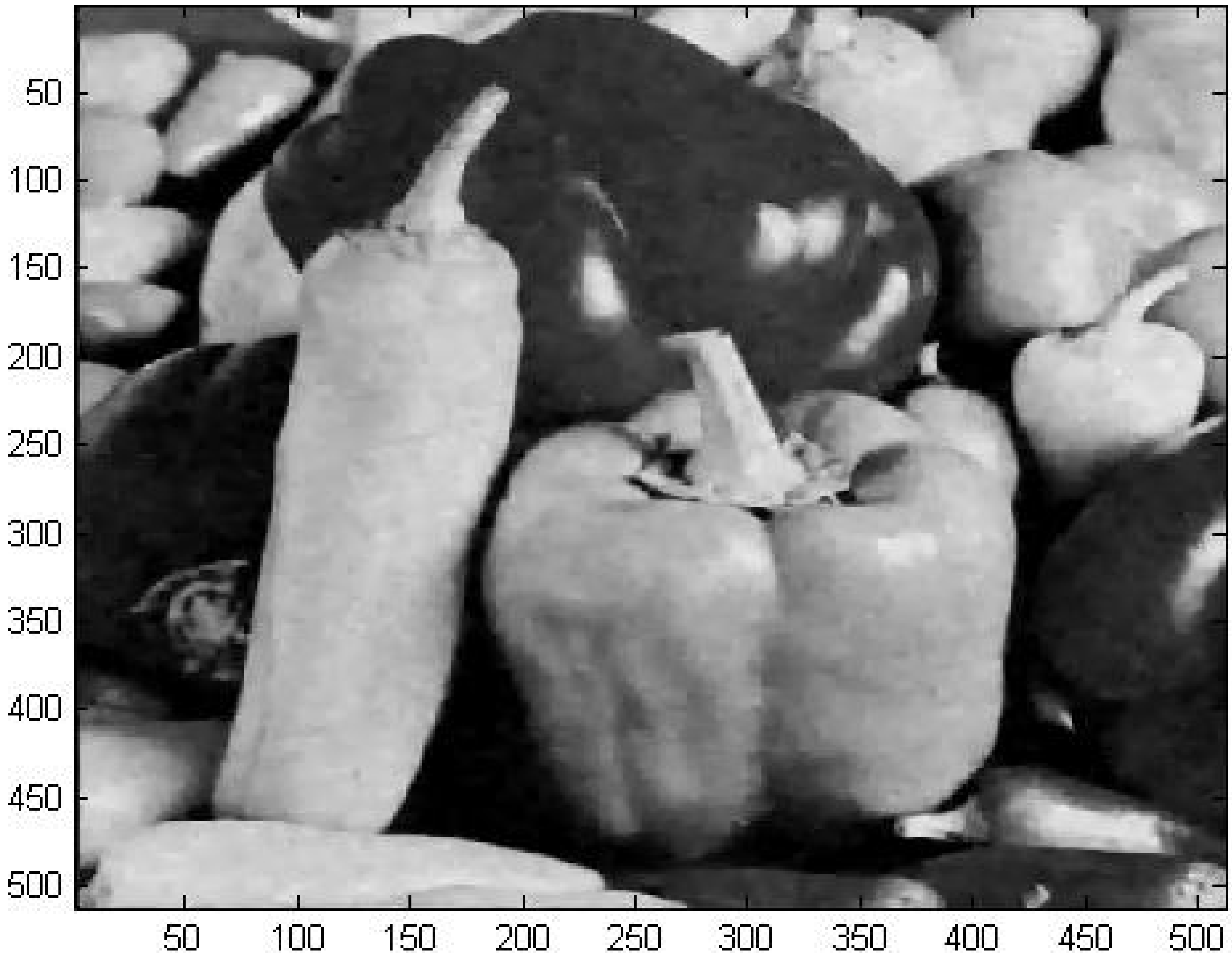


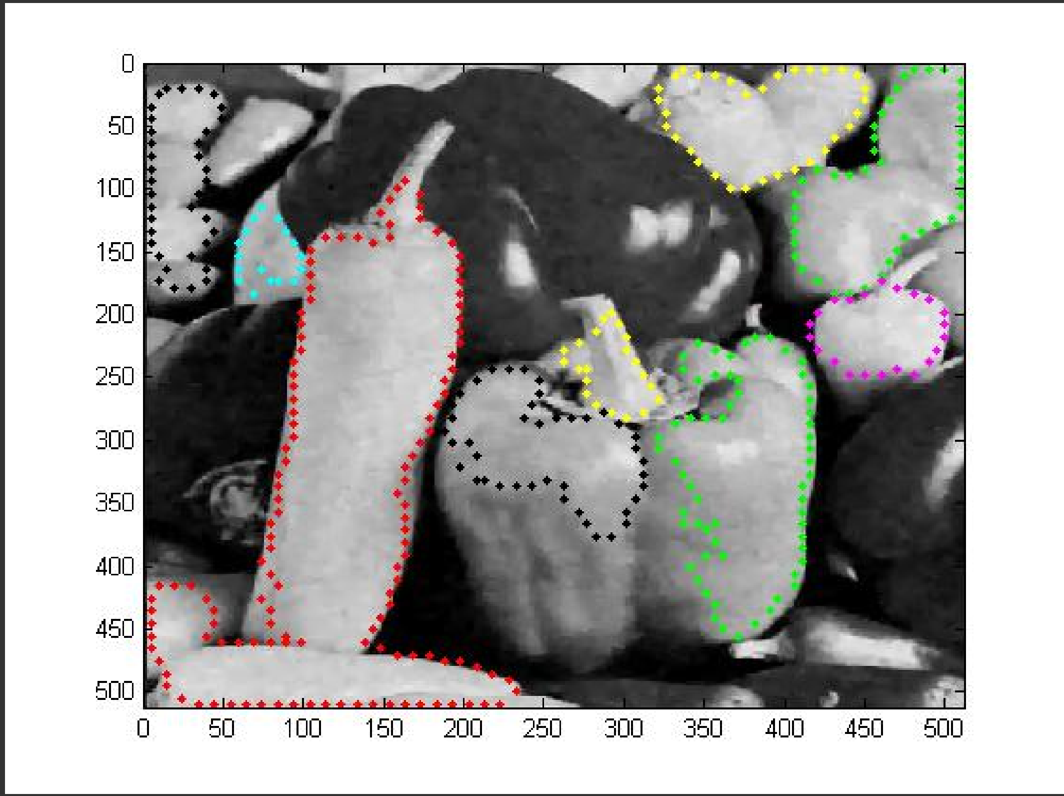

Resulting regions from both thresholds

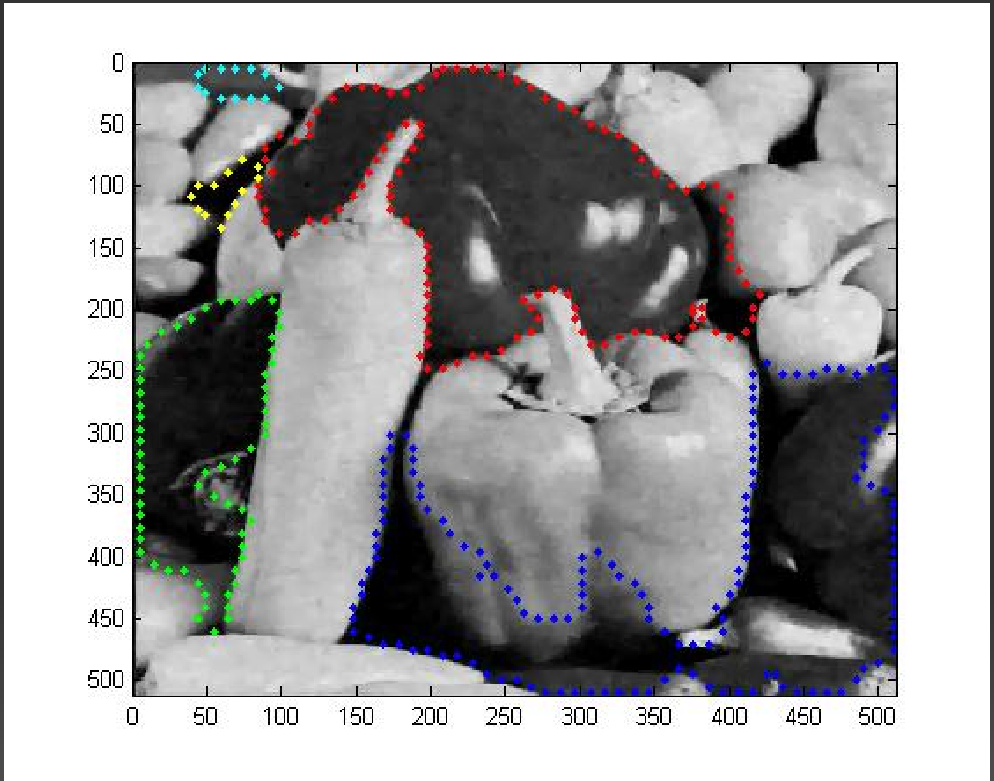

0
50
100
150
200
250
300
350
400
450
500
0
50
100
150
200
250
300
350
400
450
500


0
50
100
150
200
250
300
350
400
450
500
0
50
100
150
200
250
300
350
400
450
500



50
100
150
200
250
300
350
400
450
500
50
100
150
200
250
300
350
400
450
500



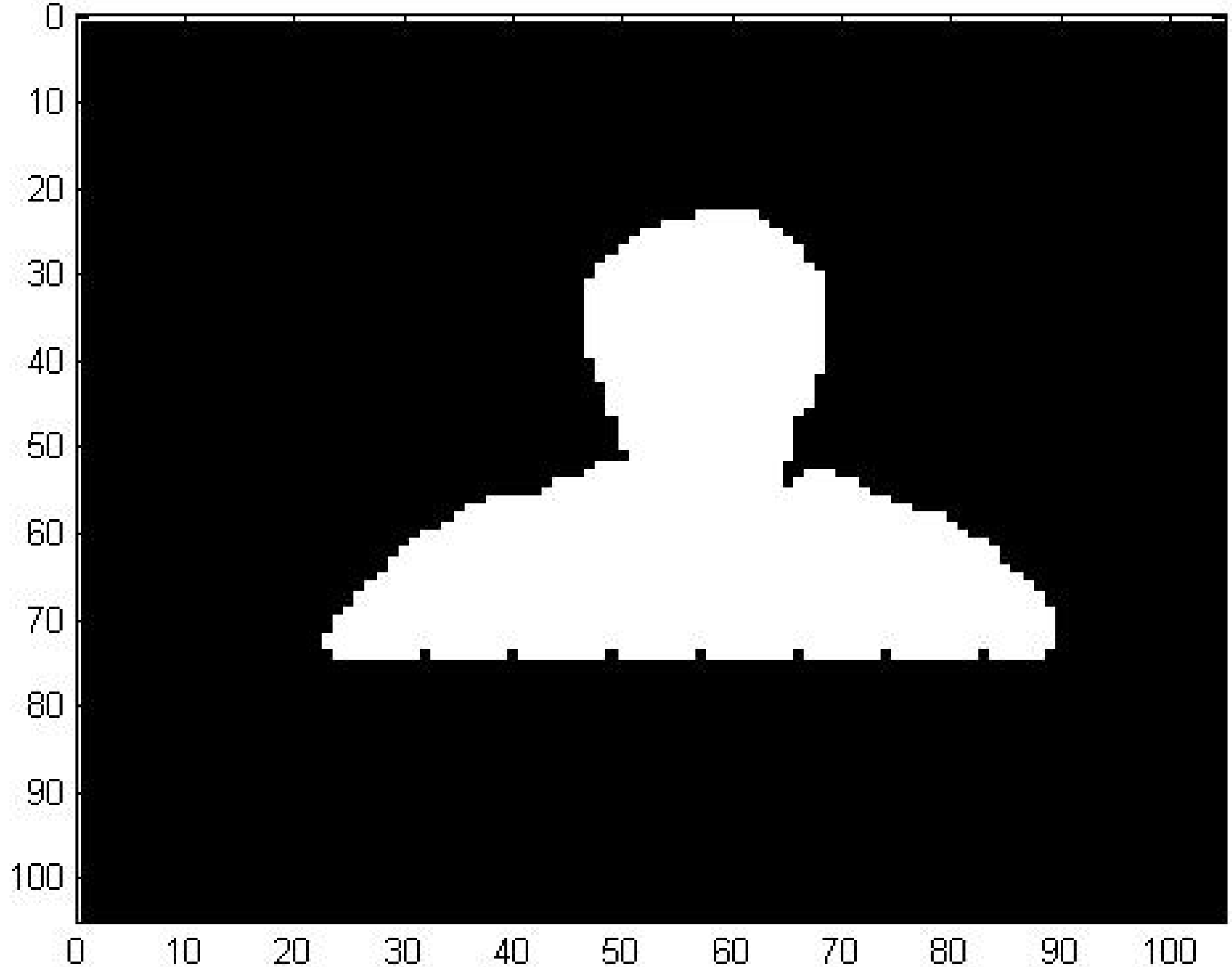

Resulting regions

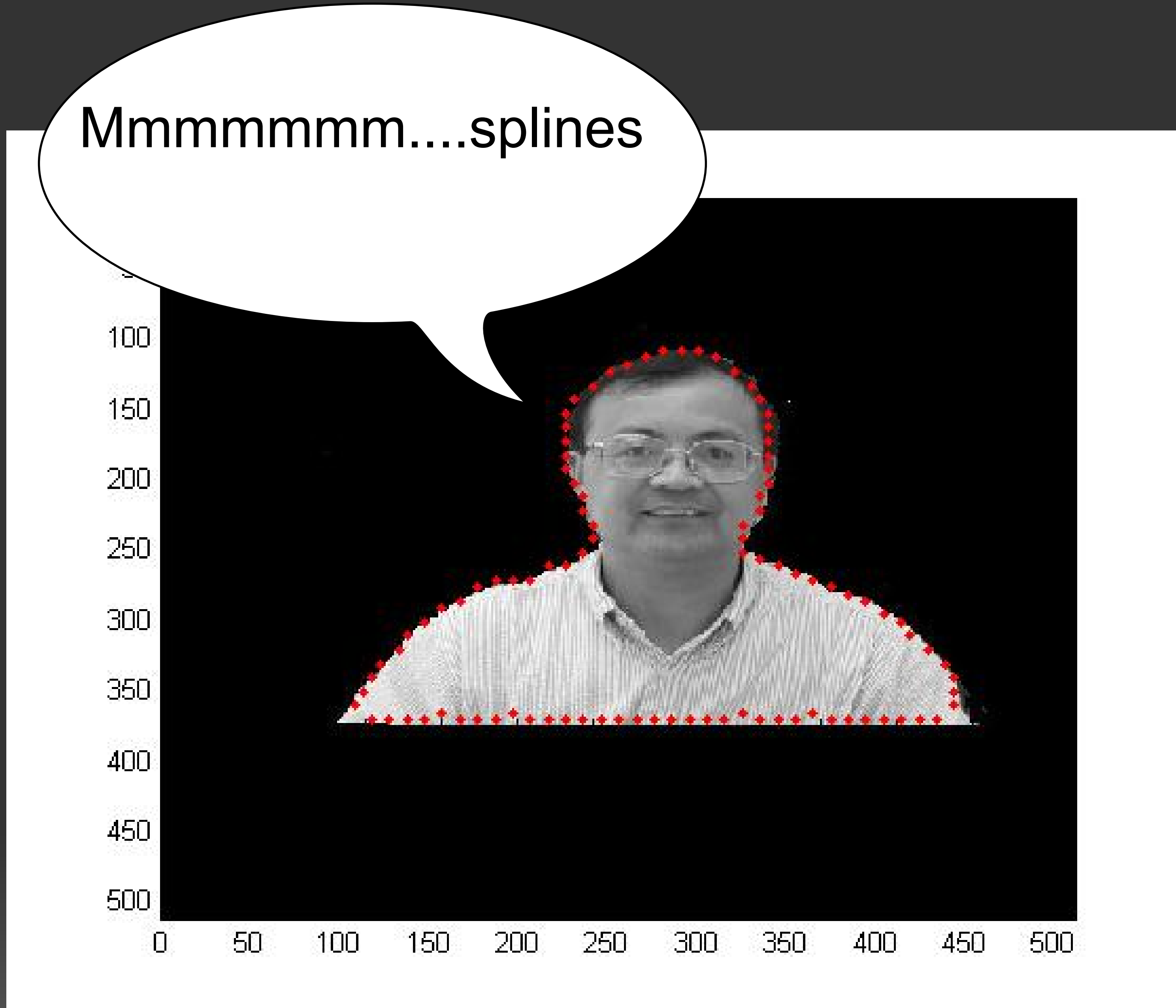
Demonstration of our method on *F-16* photo, extra-noisy *Lena*

# What more can be done

- Better thresholds (completely automatic)
- Better **clean**
- Circles, as opposed to blocks
- Different block size
- Color images
- 3-dimensions

# Who is it?






Thank you, Dr. Lai!


Edges are
tough.