# Incorporating Flexibility into the Normalized Cut Image Segmentation Algorithm

*May 02, 2012*

*Author:*
**Yi Xiang Chong**
B.S. in Mathematical Sciences
Economics-Statistics
Carnegie Mellon University

**Abstract:** Image segmentation is the process of dividing a digital image into individual segments which share similar visual characteristics. The Normalized Cut (NCut) algorithm is one of the commonly used graph-based approaches in image segmentation. The NCut algorithm aims to extract big picture segments or global features of an image, a process which closely resembles how a human would approach image segmentation [1]. However, the algorithm is heavily dependent on a constant tuning parameter that is subject to arbitrary assignment prior to running the algorithm. This tuning parameter is independent of the image and indirectly specifies the level of details in the image one requires from the segmentation. Given this shortcoming, we propose a more flexible approach that introduces a local tuning parameter for each pixel over a small neighborhood in the image. We believe that the tuning parameter should represent the local variation of the features in the image in order to correctly tune the necessary components in the segmentation process. In particular, we look at improving the segmentations by introducing multiple "local-variation" tuning parameters that are adjusted to specific regions of the image. We do this through a semi-supervised method, where the regions are defined using the segmentations of the original NCut algorithm. Through our methodology, we incorporate additional local variation into tuning the algorithm without sacrificing the global features extracted by the original NCut algorithm. Results show that our methodology manages to improve the original NCut segmentations for some sample images.
**Keywords:** image segmentation, normalized cut algorithm, graph-based segmentation, flexibility, local variation, local neighborhood, semi-supervised method.

## 1 Introduction

Image segmentation is the process of dividing a digital image into individual segments which share similar visual characteristics. Some important applications of image segmentations are used in medical imaging, face recognition, fingerprint recognition and

automated machine vision. In particular, image segmentation advancements in medical imaging have helped medical practitioners to more accurately locate cancerous cells, identify lung diseases [2], diagnose patients using Magnetic Resonance Imaging (MRI), and automate gene identification [3].

There have been several developments in image segmentation algorithms in recent years. Some popular methods for image segmentation are clustering methods, histogram-based or thresholding methods, edge detection, partial differential Equation (PDE) methods, region growing methods, artificial neural networks and graph-based methods [3,4]. Among these different approaches, graph-based segmentation methods provide an intuitive framework for image segmentation. Graph-based segmentation methods are grounded on spectral graph theory, a mathematical field that has been well established and backed with strong mathematical rigor [5]. Furthermore, graph-based methods provide an intuitive framework representing pixels as nodes in an image and the similarity as edges.

In this paper, we will explore improving the Normalized Cut Algorithm, a commonly used graph-based image segmentation algorithm. The Normalized Cut Algorithm uses a global criterion, the *normalized cut*, to segment global features of a graph-based image. In this paper, the *global features* of an image refer to the big picture segmentations, whereas *local features* of an image refer to the more detailed segmentations. For example, for an image of two people sitting on the beach, the global features of that image might be the individuals, the sand, the sea and the sky, whereas the local features of that image might be their clothes, sunglasses, and their drinks. First, we introduce the Normalized Cut algorithm and briefly discuss its theory and implementation. Second, we propose incorporating a "local-variation" tuning parameter for the individual pixels in the image that will add flexibility to the Normalized Cut algorithm. Third, we will discuss how we incorporate our "local-variation" tuning parameters into the Normalized Cut algorithm. Fourth, we evaluate the segmentation results of the more flexible Normalized Cut algorithm. Fifth, we will asses the robustness of both the original and our proposed flexible Normalized Cut algorithms on noisy images. Last, we will discuss our results and close with future work.

## 2 The Normalized Cut (NCut) Algorithm

The Normalized Cut (NCut) Algorithm is a graph based segmentation algorithm proposed by Jianbo Shi and Jitendra Malik in the year 2000. The original NCut Algorithm paper can be found at `http://www.cs.berkeley.edu/~malik/papers/SM-ncut.pdf`. The motivation behind the NCut algorithm was to extract the global features of an image, rather than focusing on the local features and their consistencies in the image data [1].

In this section, we introduce some graph theoretic definitions and the Normalized Cut criterion, and then present the Recursive 2-way NCut Algorithm. This algorithm requires the input of a graph with weight edges computed from the image. After a summary of the MATLAB implementation, we demonstrate the original NCut algorithm on an example image. Last, we discuss the algorithm's performance given the choice of tuning parameters.

### 2.1 The Normalized Cut Criterion

Let $G = (V, E)$ be an undirected weighted graph with a set of vertices $V$ and a set of unordered pairs of edges $E$, where each $(i, j) \in E$ has an associated weight $w_{ij}$. The weight, $w_{ij}$ indicates the strength of the connection, or edges, between node $i$ and $j$. A strong connection between node $i$ and $j$ would have a high $w_{ij}$ value, and vice versa. Strong connections between a group of nodes generally indicates that these nodes are very similar and belong as one cluster.

The graph $G$ can be segmented into two disjoint subsets of the graph, $G_1$ and $G_2$, if we remove the connecting edges between the sets $G_1$ and $G_2$. In graph theory, the removal of these edges, or a *cut*, is a partition of graph vertices into two disjoint subsets. The cut of a weighted graph $G$ into two disjoint subsets $G_1$ and $G_2$ is defined as:

$$cut(G_1, G_2) = \sum_{i \in G_1, j \in G_2} w_{ij} \tag{1}$$

Generally, two different graph clusters with weak connections between them will have a small cut value if we partition them. Usually, the goal of segmentation is to find a set of clusters that correspond to low cut values. The *degree* of a vertex, or

node, in a weighted graph is the total weight of the edges incident to the node. Let $d_i$ be the degree of a node $i$ in the graph. Then:

$$d_i = \sum_j w_{ij} \tag{2}$$

The *volume* of a subset $G_1$ of a weighted graph represents how dense the subset $G_1$ is in terms of its edge weights. In general, a high volume value indicates that the connections within the subset are strong. Let $vol(G_1)$ be the volume of the subset $G_1$. Then:

$$vol(G_1) = \sum_{i \in G_1} d_i \tag{3}$$

Then, the *Normalized Cut* criterion [1] is given as:

$$NCut(G_1, G_2) = \frac{cut(G_1, G_2)}{vol(G_1)} + \frac{cut(G_1, G_2)}{vol(G_2)} \tag{4}$$

The Normalized Cut Algorithm minimizes the *Normalized Cut*(NCut) criterion. In general, for two different graph clusters, strong internal connections within the clusters indicate similar grouped nodes and weak connections between these clusters indicate that these two clusters are different. Intuitively, by minimizing the normalized cut criterion for groups $G_1$ and $G_2$, we find a *cut* such that the connections between the newly partitioned groups are weak and that the nodes are evenly distributed so that the internal connections for the new groups are both evenly strong. By minimizing the normalized cut criterion for $G_1$ and $G_2$, we try to find new balanced partitions that gives a small cut value and strong internal connections for both the partitions at the same time [6].

In solving this minimization problem in (4), let $D$ be an $N$ x $N$ diagonal matrix with degree $d$ on its diagonal. Let $W$ be an $N$ x $N$ symmetrical matrix with $W(i, j) = w_{ij}$. The $W$ matrix is known as the *affinity matrix*. Nodes with high affinity will have high weight values, while nodes with low affinity will have low weight values.

Jian and Malik showed that the minimization problem of the NCut criterion in (4) can be solved by solving the following generalized eigenvalue system:

$$(D - W)x = \lambda Dx \tag{5}$$

The second smallest eigenvector of the generalized eigensystem (5) is the real valued solution to the NCut problem. Thus, we solve the eigensystem (5) for eigenvectors and use the eigenvector with the second smallest eigenvalue to segment our graph or image.

## 2.2 The Recursive Two-way NCut Algorithm

Let the image we are trying to segment be the set of pixels $I$.

1. Given the set of features, construct a weighted graph $G = (V, E)$, compute the edge weights $W(i, j)$. In this application, the image $I$ is the graph, and the pixels are the nodes. Calculate the corresponding $D$ matrix.
2. Solve $(D - W)x = \lambda Dx$ for eigenvectors with the smallest eigenvalues.
3. Select the eigenvector with the *second smallest* eigenvalue to bipartition the graph by finding the splitting point such that the NCut criterion is minimized.
4. Decide if the current partition should be subdivided by checking the stability of the cut, and make sure that NCut is below the prespecified threshold.
5. Recursively repartition the segmented parts if necessary given the number of segments specified by the user.

## 2.3 Constructing the Affinity Matrix

We need to define the edge weights $w_{ij}$ prior to starting the recursive two-way NCut Algorithm. In the original NCut algorithm, each entry $w_{ij}$ in the affinity matrix $W$ is constructed as follows [1]:

$$w_{ij} = e^{\frac{-\|F(i)-F(j)\|_2^2}{\sigma_F}} * \begin{cases} e^{\frac{-\|X(i)-X(j)\|_2^2}{\sigma_X}} & \text{if } \| X(i) - X(j) \|_2 < R \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

where, $X(i)$ is the spatial location of node i

$F(i)$ is a feature vector based on intensity, color, or texture information of node i

$\sigma_F, \sigma_X$ are feature and spatial tuning parameters respectively

For a black and white image, the feature (sometimes called intensity) of a pixel $i$, $F(i)$, takes values between 0 (black) and 255 (white). Note that, the weight $w_{ij} = 0$ for any pair of pixels $i$ and $j$ that are lying more than $R$ pixels apart.

The parameter $\sigma_F$ acts as a tuning parameter that controls how much of the magnitude of the difference between features $\parallel F(i) - F(j) \parallel$ are incorporated into computing the edge weights. Since we are dividing $- \parallel F(i) - F(j) \parallel$ by $\sigma_F$, a *smaller* $\sigma_F$ value decreases the affinity/weight values $w_{ij}$, thus resulting in less "tightly grouped" pixels and a more *local* segmentation, and vice versa for high $\sigma_F$ value (examples in Section 2.6). Similarly, $\sigma_X$ acts as a tuning parameter for the degree of spatial features $\parallel X(i) - X(j) \parallel$ that are used to compute the edge weights. In Jianbo and Malik's implementation of the NCut Algorithm in MATLAB, the default values for the parameters are $\sigma_F = 0.1$, $\sigma_X = 0.3$, and $R = 10$. However, note that these parameters $\sigma_F$, $\sigma_X$ are constant and applied to all features in the image regardless of the feature values. In this paper, we will instead focus on a more flexible local tuning parameter $\sigma_{ngbhd}$ that hopefully will improve the segmentation.

## 2.4 Implementation in MATLAB

The creators of the algorithm, Malik and Shi, have generously made their implementation of the original NCut Algorithm in MATLAB available online at `http://www.cis.upenn.edu/~jshi/software/`. In this research, we will be using their implementation of the NCut Algorithm in MATLAB and focus on introducing flexibility to the construction of the affinity matrix. On the next page, Figure 1 is a summary of how their implementation of the NCut Algorithm works on images, with increased focus on the construction of the affinity matrix.

The NCut algorithm first reads in an image of size $m$ x $m$ and constructs an intensity matrix, $I$, of size $m$ x $m$ corresponding to the pixels in the image. As an illustrative example, assume we have a black and white image. Then, the intensity matrix, $I$, consists of the feature values, or intensity values of the pixels, ranging from 0 (black) to 255 (white). The intensity matrix $I$ is then input into a MATLAB function called "ICGraph". The ICGraph function does the following: (i) calculates the edges of the image through the "computeEdges" function and returns the 'EdgeMap' of the image; (ii) computes the affinity matrix of the image, $W$, through the "computeW" function using the 'EdgeMap' profile of the image.

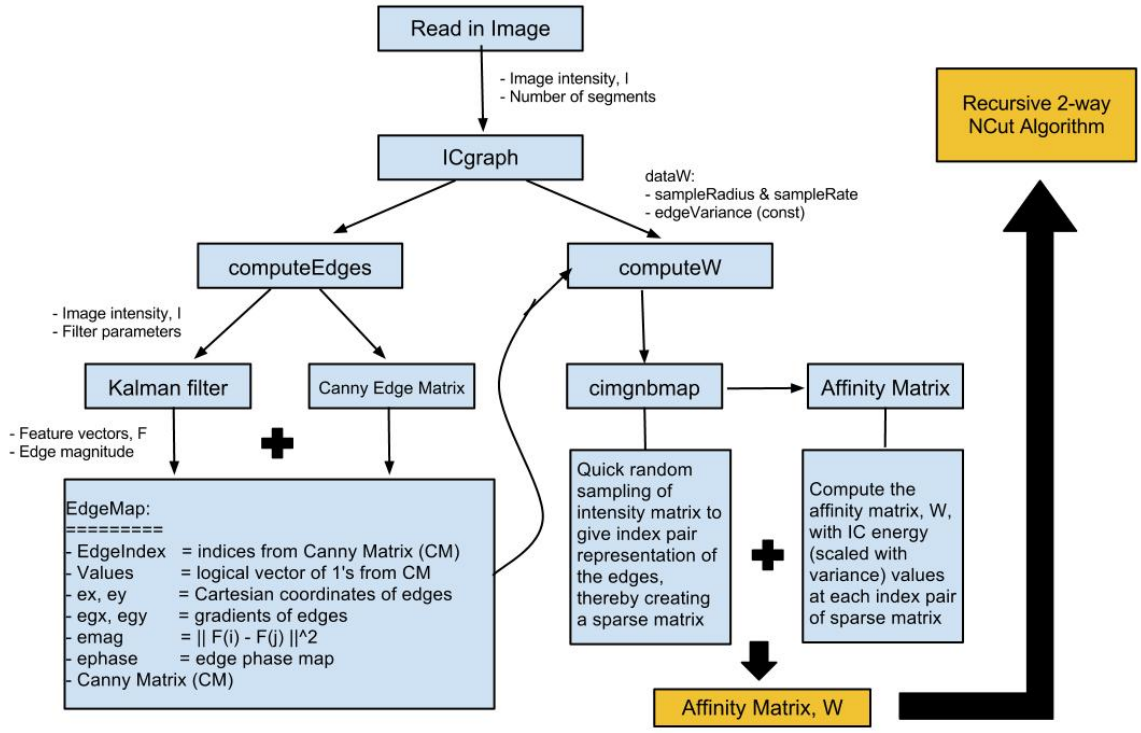


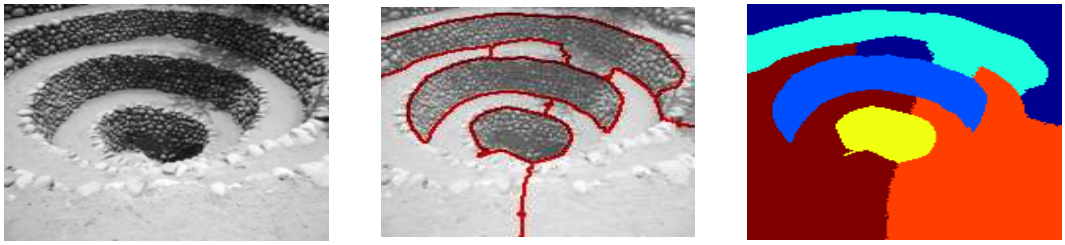

Figure 1: Summary of The NCut Algorithm Implementation with Focus on Affinity Construction

In the "computeEdges" function, the algorithm uses the Canny Edge Detector and a Kalman Filter [7] to find the edges or contours of the image. The Canny Edge Detector [8] is an algorithm that returns the edges in the image (more details on how it works can be found in Section 3.3.1). The Kalman Filter is then used to filter out the noise in the edges returned by the Canny Edge Detector to get smoother and more accurate edges. With the smoother edges, the algorithm then builds an 'EdgeMap' profile that describes the gradients and coordinates of the edges in the image and the magnitude of the pixel features. In the "computeW" function, the algorithm takes in the 'EdgeMap' profile of the image and through the "cimgnbmap" function, does a quick random sampling on the intensity matrix to create a sparse matrix of index pairs representation of the edges according to the spatial component constraint in (6). This sparse matrix of index pairs, together with the magnitude of the pixel features in the 'EdgeMap' object, is then used to compute the affinity matrix tuned by the $\sigma_F$ tuning parameter (refered as 'edgeVariance' in Figure 1) (Recall Equation (6)). Finally, the Affinity Matrix, $W$, is fed into the Recursive two-way NCut Algorithm

described in Section 2.2.

## 2.5 The Original NCut Algorithm On An Example Image: The Well Image

We run the original NCut Algorithm on an example image to illustrate its behavior and performance. This example is a black and white image of a spiraling well from our personal image repository. This well image has three height levels to it and tiny little rocks between these levels. Then, these three levels are considered as the global features and the tiny little rocks as local features of the image. This well image will be the running illustrative example for the paper.



(a) A spiraling well (b) The segmentation with lines (c) The segmentation in color

Figure 2: (a) The "spiraling well" image example; (b)-(c) The segmentations of the well image, with lines and in color, after running the original NCut algorithm with *Segments* = 6, $\sigma_F = 0.1$, $\sigma_X = 0.3$, $R = 10$.

We chose six segments for the segmentation just for illustrative purposes. We see similar segmentations with different number of segments as well. From Figure 1(b), as expected from the NCut criterion, we can see that the original NCut algorithm overall gives big picture, global segmentation. In this case, instead of focusing and segmenting out the tiny little rocks in the image, the algorithm succeeds at segmenting out the group of little rocks as one whole group.

However, the algorithm is only somewhat successful at correctly grouping each of the levels of the spiraling well image. Note that, the spiraling well image has three *levels* to it, and there is a shadow of a tree (on the right of the image) cast upon the different levels of the well. The original algorithm did not manage to correctly segment the 2nd and 3rd level of the well image. First, one can see clearly from the colored segmentation in Figure 1(c), the 2nd and 3rd level have been treated as one group and then vertically segmented into half as two separate regions (*brown* and

*orange*). This is obviously incorrect by inspection. Second, from Figure 1(b), there is an additional segment (*dark blue* in Figure 1(c)) included in the segment of the 2nd level of the well image. Third, from Figure 1(c), the algorithm incorrectly segments the *teal* colored group of rocks. In this *teal* colored segment, part of the tree shadow was incorrectly grouped with the rocks in the segment.

For this well example, a reasonable segmentation would be to correctly segment out each of the individual levels of the well as a whole. In summary, the original NCut algorithm succeeds at giving a global segmentation but gives somewhat *reasonable* segments that are not sensitive at picking up local variation in the image. We try to add flexibility and improve its performance by constructing the affinity matrix through local tuning. With our proposed methodology, we address the problems that arose from the original NCut segmentations on the well image and give a more reasonable and flexible segmentation of the well image.

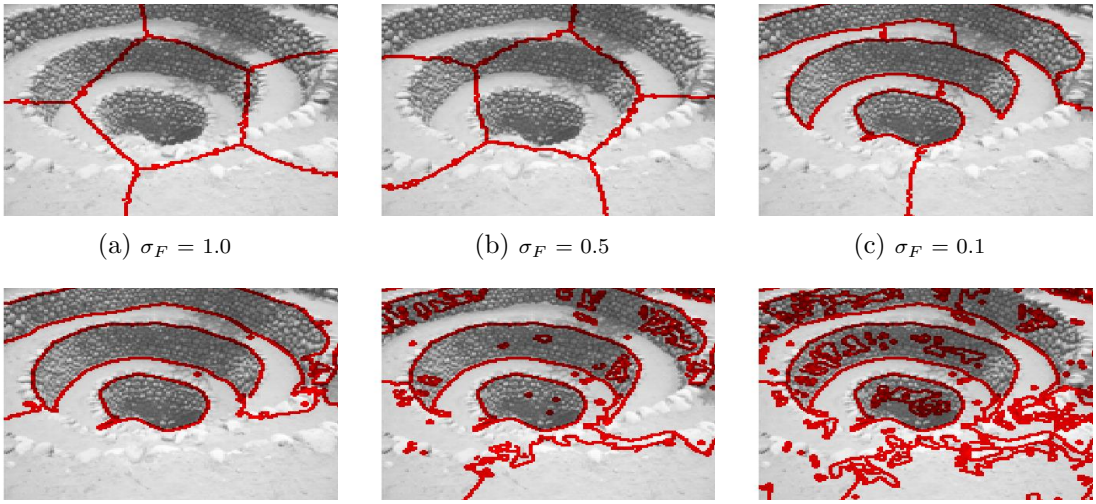
## 2.6 Varying Constant Values of $\sigma_F$

The previous segmentation result in Figure 2(b) and (c) used a fixed and pre-determined $\sigma_F$ value. We first tried varying different constant values of $\sigma_F$ to see how they affect the segmentations. In particular, we would like to see (i) how sensitive the segmentations are to different constant values of $\sigma_F$; and (ii) how well these constant values perform on the segmentation process for the well image.

As mentioned before in Section 2.3, when constructing the affinity matrix, since we are dividing $- \parallel F(i) - F(j) \parallel$ by $\sigma_F$, a *lower* $\sigma_F$ value means we want more of the magnitude of the image features to be taken into account when computing the edge weights, thus resulting in less tightly grouped pixels and a more *local* segmentation, and vice versa for high $\sigma_F$ value.

Recall that in the original NCut algorithm, $\sigma_F = 0.1$. We tried a range of $\sigma_F$ values and present results for 1.0, 0.5, 0.1, 0.05, 0.01 and 0.005. The segmentations for the image stayed the same for values of $\sigma_F$ below 1.0 and above 0.005. The segmentations are shown in Figure 3 on the next page.

From Figure 3, one can see as the $\sigma_F$ decreases from 1.0 to 0.01, the segmentations become more local (more detailed) and less global (less detailed). For example, at the



(a) $\sigma_F = 1.0$ (b) $\sigma_F = 0.5$ (c) $\sigma_F = 0.1$

(d) $\sigma_F = 0.05$ (e) $\sigma_F = 0.01$ (f) $\sigma_F = 0.005$

Figure 3: (a)-(f) Segmentation results for various fixed $\sigma_F$ values

extreme ends, when $\sigma_F = 1.0$, the image is segmented into large unstructured segments, and when $\sigma_F = 0.01$, the segmentation is more detailed in that the algorithm picked up the tiny little rocks in the image. However, for the extreme $\sigma_F$ values of 1.0 and 0.01, the segmentations are, respectively, either too global or too detailed. Interestingly, when $\sigma_F = 0.05$, the original algorithm gives a reasonable segmentation of the well image. We see that the algorithm is sensitive to the values of $\sigma_F$, and different values give reasonable segmentations in different parts of the image. Given that $\sigma_F$ is fixed, we seek to add flexibility to the algorithm by automatically finding the appropriate $\sigma_F$ values for specific regions in the image using the local variation of the image features. Furthermore, we would like to have individual $\sigma_F$ values for different areas in the image to further localize and better fine tune the original segmentation result.

# 3 A More Flexible Approach: A Shape-Based "Local-Variation" Tuning Matrix

In this section, we propose a more flexible way to construct the affinity matrix of an image that might improve the segmentation result of the original algorithm. In the following subsections, we will give the various approaches that we have tried leading to our final methodology.

To serve as a general overview of how our final methodology works, we will provide a quick discussion of the intuition behind it. Notice that the constant $\sigma_F$ parameter in Equation (4) is applied to all values of $\| F(i) - F(j) \|$ and the parameter is independent of the features $F(i)$, $F(j)$ at nodes $i$, $j$ in the image $I$.

Unfortunately, the constant value $\sigma_F$ may not appropriately scale the magnitude of the difference between features $\| F(i) - F(j) \|$ in the computation of the edge weights. We think $\sigma_F$ will serve as a better tuning parameter if we incorporate local tuning parameters in different areas of the image and obtain local information about the image. Hence to achieve that, for *each* pixel, we introduced a new $\sigma_{ngbhd}$ tuning parameter as a function of the local variation of the features $F$ in the image. We believe a more accurate description and structure of the edge weights can be achieved if the magnitude of the difference between features are tuned according to the local variation of the features themselves [9]. Furthermore, having a specific local tuning parameter for each pixel allows for self tuning in the affinity matrix without the need of selecting a single constant $\sigma_F$ value [9].

By doing this, we now have varying values of $\sigma_{ngbhd}$ for each pixel. As we will see later in the image example in Section 3.2, having too many different values of $\sigma_{ngbhd}$ might give a "patchy" and over local segmentation. Hence, to smooth out the patchy segmentation, for particular regions of interest in the image, we set the $\sigma_{ngbhd}$ value to be a single constant value which is the most frequently occurred mode $\sigma_{ngbhd}$ value for each of those regions. We believe the mode value for a particular region may best describe the appropriate local constant $\sigma_{ngbhd}$ value. Most importantly, by introducing the use of regions, we believe that we can add flexibility to the NCut algorithm to customize and tune the edge weights according to specific image regions of interest. One natural way to choose these regions of interest is to use the segments from the original NCut algorithm.

In summary, to add flexibility to the NCut algorithm and to improve the original segmentation, we proposed the following methodology: we first introduce 'local-variation' tuning parameters, $\sigma_{ngbhd}$, as a function of the local variation of the neighborhood features around each pixel; then second, smooth these varying values of $\sigma_{ngbhd}$ by taking the mode of $\sigma_{ngbhd}$ values of particular regions in the image; third,

apply these single constant mode values of each region accordingly to the computation of the weight edges. We believe by doing that, we are able to better tune the local features when computing the edge weights. These multiple parameters that are functions of the local variation of the features $F$ add flexibility to the NCut algorithm are constant for specific regions of the image to preserve some aspects of the big picture.

## 3.1 "Local-Variation" Tuning Parameter

As mentioned before, one problem with the current implementation of the $\sigma_F$ tuning parameter is its lack of dependence on the local features of the image $I$. Recall from the spiraling well image in Figure 1(b), the original NCut algorithm gave unreasonable segments. We believe this may be due to the global tuning parameter, $\sigma_F$'s, lack of dependence on the local variation of the image features. In a way, the scale parameter must "understand" the local variation of the features in the image in order to correctly tune the feature difference between two pixels. We try to improve this by introducing a "local-variation" tuning parameter for each pixel $i$, $\sigma_{ngbhd}(F(i), r)$. This new tuning parameter, $\sigma_{ngbhd}(F(i), r)$ is defined as the standard deviation of the neighborhood features around pixel $i$ of radius $r$. For an image $I$ of size $m$ x $m$, the local-variation tuning parameter for any pixel $i$ with position index $(p, q)$ in the image matrix of size $r = 1$, $\sigma_{ngbhd}(F(i), r = 1)$, is defined as follows:

$$\sigma_{ngbhd}(F(i), r=1) = \sqrt{var \begin{bmatrix} \ddots & \vdots & \iddots \\ & F(i-m-r)_{p-r,q-r} \quad F(i-m)_{p-r,q} \quad F(i-m+r)_{p-r,q+r} & \\ \cdots & F(i-r)_{p,q-r} \quad \mathbf{F(i)_{p,q}} \quad F(i+r)_{p,q+r} & \cdots \\ & F(i+m-r)_{p+q-r} \quad F(i+m)_{p+r,q} \quad F(i+m+r)_{p+r,q+r} & \\ \iddots & \vdots & \ddots \end{bmatrix}}$$

$$for \quad i = 1, \ldots, m^2 \quad \text{and} \quad some \quad p, q \in \{1, \ldots, m\} \tag{7}$$

Note that, by definition, $var_{ngbhd} = \sigma^2_{ngbhd}$.

For example, let $I$ be a 5 x 5 image. This image contains a total of 25 pixels with each pixel $i$ labeled from $i = 1, \ldots, 25$ starting from the top left pixel and ending at the bottom right pixel. For instance, the first pixel of the image, $i = 1$, will be

located at position (1,1) in the image matrix, and the last pixel of the image, $i = 25$, will be located at position (5,5) in the image matrix. Each pixel $i$ has features, $F(i)$, associated with it. Then, the feature neighborhood of radius 2 ($r = 2$) of 13, $F(13)$, is illustrated as follows:

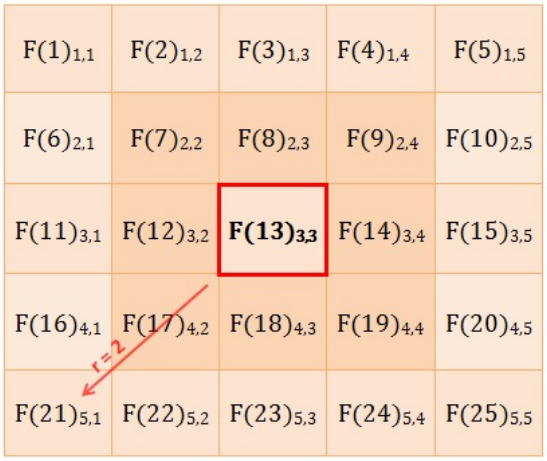



Figure 4: Neighborhood of radius 2 of the feature of pixel 13, F(13)

### 3.1.1 Computing $var_{ngbhd}(F(i), r)$

There are several ways to compute $var_{ngbhd}(F(i), r)$. In the following subsection, we define two methods. The first is to have *equal weights on the neighborhood rings* in the variance and the second is to have *exponential weights as functions of the radius of the different neighborhood rings* in the variance.

We will use two neighborhood rings as an example for this subsection, but note that the same principles hold for using $r$ rings. We first define the following:

$$Ring\ 1 = \begin{bmatrix} \ddots & & & & & \iddots \\ & * & * & * & * & * & \\ & * & F(i-m-1)_{p-1,q-1} & F(i-m)_{p-1,q} & F(i-m+1)_{p-1,q+1} & * & \\ & * & F(i-1)_{p,q-1} & \mathbf{F(i)_{p,q}} & F(i+1)_{p,q+1} & * & \\ & * & F(i+m-1)_{p+1,q-1} & F(i+m)_{p+1,q} & F(i+m+1)_{p+1,q+1} & * & \\ & * & * & * & * & * & \\ \iddots & & & & & \ddots \end{bmatrix},$$

$$Ring\ 2 = \begin{bmatrix} \ddots & & & & & \iddots \\ & F(i-2m-2)_{p-2,q-2} & F(i-2m-1)_{p-2,q-1} & F(i-2m)_{p-2,q} & F(i-2m+1)_{p-2,q+1} & F(i-2m+2)_{p-2,q+2} \\ & F(i-m-2)_{p-1,q-2} & * & * & * & F(i-m+2)_{p-1,q+2} \\ & F(i-2)_{p,q-2} & * & \mathbf{F(i)_{p,q}} & * & F(i+2)_{p,q+2} \\ & F(i+m-2)_{p+1,q-2} & * & * & * & F(i+2m+2)_{p,q+2} \\ & F(i+2m-2)_{p+2,q-2} & F(i+2m-1)_{p+2,q-1} & F(i+2m)_{p+2,q} & F(i+2m+1)_{p+2,q+1} & F(i+2m+2)_{p+2,q+2} \\ \iddots & & & & & \ddots \end{bmatrix}$$

$$\bar{F} = \text{grand mean of neighborhood features} = \frac{1}{n}\left[\sum_{\substack{F(i)\in \\ Ring\ 1\ \cup\ Ring\ 2}} F(i)\right], \quad \text{where } n = |Ring\ 1 \cup Ring\ 2|$$

Note that the union of Ring 1 and Ring 2, Ring 1 ∪ Ring 2, is just the neighborhood of radius 2 ($r = 2$) of the feature of pixel $i$. Compared to Figure 4, the union of these two rings is just another illustration of a neighborhood for $r = 2$ for the feature of any pixel $i$ in a $m$ x $m$ sized image.

**Equal Weights on the Neighborhood Rings**

For any pixel $i$, we let $var_{ngbhd}(F(i), r)$ have equal weight components. For example, for an image $I$ with size $m$ x $m$, for $r = 2$, the parameter $var_{ngbhd}(F(i), r = 2)$ with equal weight components is defined as follows:

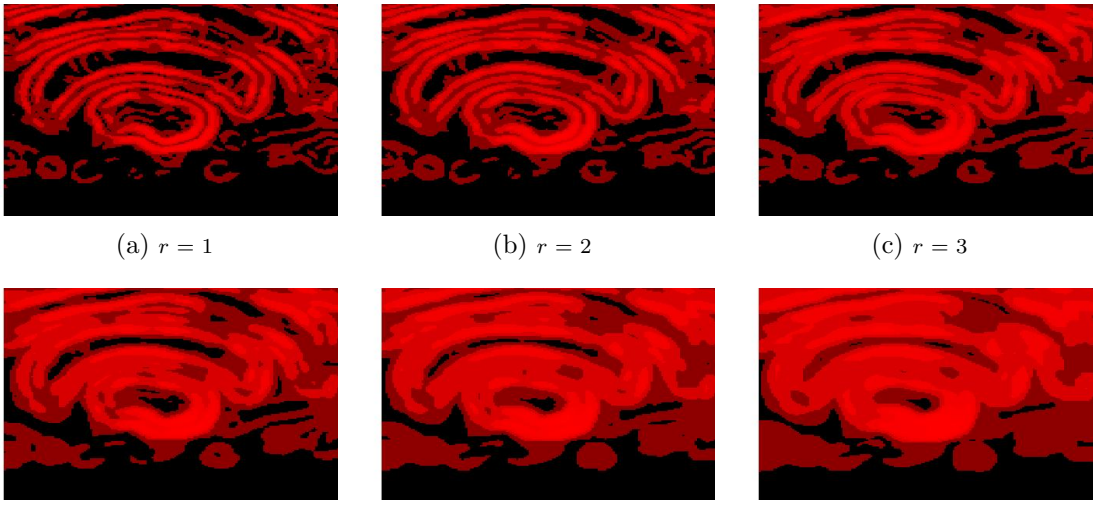
$$var_{ngbhd}(F(i), \{r = 2\}) = \underbrace{\frac{|Ring\ 1|}{n-1}}_{w_1} \sum_{F(i)\in Ring\ 1} (F(i) - \bar{F})^2 \quad + \quad \underbrace{\frac{|Ring\ 2|}{n-1}}_{w_2} \sum_{F(i)\in Ring\ 2} (F(i) - \bar{F})^2$$

$$where \quad w_1 + w_2 = \frac{n}{n-1} \tag{8}$$

We choose the weights $w_1$, $w_2$ such that the pixels will have the same total weight as they would in the sample variance, $\frac{n}{n-1}$. To clearly visualize the local-variation $\sigma_{ngbhd}$ values of an image for a particular $r$ using equal weights on the neighborhood rings, we present a *heatmap* of the equal weighted neighborhood rings $\sigma_{ngbhd}$ values of the well image. A heatmap is a graphical representation of the data values relative

to themselves indicated by varying contrast of a color (usually, red color is used to signify 'heat'). For example, a darker red color in a specific region indicates that the values in that region are relatively lower, and a brighter red color in another region indicates that the values in that region are relatively higher.

We present different heatmaps of the equally weighted $\sigma_{ngbhd}$ values for the well image for different neighborhood radius values, $r$ = 1,2,3,4,5,6 in Figure 5 below:



(a) $r = 1$ (b) $r = 2$ (c) $r = 3$

(d) $r = 4$ (e) $r = 5$ (f) $r = 6$

Figure 5: (a)-(f) Heatmaps of the equal weighted neighborhood ring $\sigma_{ngbhd}$ values for $r = 1$ to 6.

As one can see from Figure 5, as $r$ increases from 1 to 6, more regions in the different heatmaps have increasingly brighter red color. As the neighbohood radius $r$ increases, we have higher $\sigma_{ngbhd}$ values in general. As the neighborhood $r$ increases, the neighborhood size for the computation of $\sigma_{ngbhd}(F(i), r)$ for each pixel $i$ increases, which then allows $\sigma_{ngbhd}(F(i), r)$ to detect local neighborhood variation in features extending further around each pixel $i$. Since this well image has varying feature intensity for different well levels, the equally weighted $\sigma_{ngbhd}$ value increases for each pixel. In particular, as the neighborhood radius $r$ increases, one can see the increasingly bright red color on the contour lines of the well levels as each $\sigma_{ngbhd}$ value on the contour lines manages to capture the variation of the different feature intensities across the different levels of the well. We are also less able to distinguish the different shapes of the image with high $r$ values.

**Exponential Weights as Functions of the Radius of the Different Neighborhood Rings**

To further fine tune our local-neighorhood parameter, we let $var_{ngbhd}(F(i), r)$ have exponential weight components on the rings. We want the neighborhood features closer to pixel $i$, in a decaying fashion, to be given more importance/weight in the variance computation. We would like $var_{ngbhd}(F(i), r)$ to be more sensitive towards the neighborhood features closer to pixel $i$.

For example, for an image $I$ with size $m$ x $m$, for $r = 2$, the parameter $var_{ngbhd}(F(i), \{r = 2\})$ with exponential weight components is defined as follows:

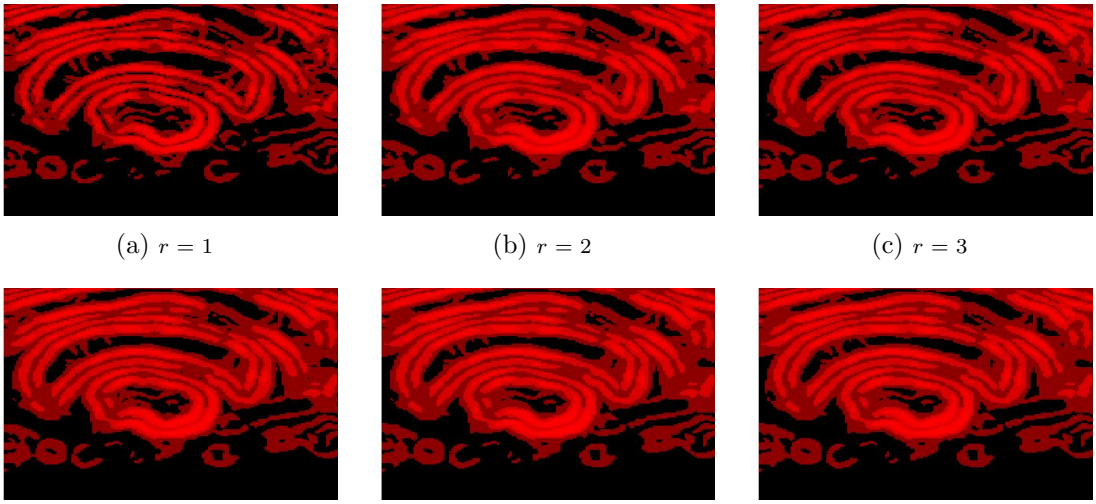
$$var_{neighborhood}(F(i), \{r = 2\}) = \underbrace{\frac{exp(-\{r = 1\})}{\sum_{i=1}^{r} exp(-i)} \cdot \frac{|Ring\ 1|}{n-1}}_{w_1} \sum_{F(i) \in Ring\ 1} (F(i) - \bar{F})^2 + \underbrace{\frac{exp(-\{r = 2\})}{\sum_{i=1}^{r} exp(-i)} \cdot \frac{|Ring\ 2|}{n-1}}_{w_2} \sum_{F(i) \in Ring\ 2} (F(i) - \bar{F})^2 \quad (9)$$

$$where \quad w_1 + w_2 = \frac{n}{n-1}$$

Note that each exponential weight is a function of the radius $r$ for the specific ring of the neighborhood. The notation $exp(-\{r = 1\})$ is meant to illustrate that the value 1 taken as an input to the exponential function is just the radius for Ring 1 of the neighborhood. We again choose the weights $w_1$, $w_2$ such that the pixels will have the same total weights as they would in a sample variance, $\frac{n}{n-1}$.

We present different heatmaps of the exponentially weighted $\sigma_{ngbhd}$ values for the well image for different neighborhood radius values, $r$ = 1,2,3,4,5,6 in Figure 6 on the next page.

In Figure 6, similar to Figure 5, as $r$ increases from 1 to 6, slightly more regions in the different heatmaps have increasingly brighter red color. However, in contrast to Figure 5, the rate at which the bright red color expands to other regions in image is slower and more contained. Since each of the rings in the neighborhood are exponentially weighted in the computation of $\sigma_{ngbhd}(F(i), r)$ for each pixel $i$, the $\sigma_{ngbhd}$ value for each pixel grows more slowly as the neighborhood size $r$ increases. In other



(a) $r = 1$ (b) $r = 2$ (c) $r = 3$

(d) $r = 4$ (e) $r = 5$ (f) $r = 6$

Figure 6: (a)-(f) Heatmaps of exponentially weighted neighborhood rings $\sigma_{ngbhd}$ values of for $r = 1$ to 6.

words, with exponentially weighted rings, outer rings are less important and increasing the neighborhood size $r$ does not have that much of an effect on increasing the $\sigma_{ngbhd}$ value. We are still able to distinguish the different shapes for higher values of $r$.

## 3.2 Incorporating the Local Tuning Parameters into the Affinity Matrix

We choose computing $\sigma_{ngbhd}$ using exponential weights on the neighborhood rings. Although the exponentially weighted $\sigma_{ngbhd}$ is statistically biased (need to explain why), hopefully by introducing bias, we can reduce the overall variance of the affinity matrix.

Let $\Sigma_{ngbhd}(r)$ be a $m$ x $m$ matrix of local-variation tuning parameters for a $m$ x $m$ image. Each entry $(p, q)$ of $\Sigma_{ngbhd}(r)$ is just $\sigma_{ngbhd}(F(m*p-m+q), r)$ . In particular, the $\Sigma ngbhd(r)$ matrix is defined as follows:

$$\Sigma_{ngbhd}(r) = \begin{bmatrix} \sigma_{ngbhd}(F(1),r) & \dots & \dots & \sigma_{ngbhd}(F(m),r) \\ \vdots & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ \sigma_{ngbhd}(F(m^2-m+1),r) & \dots & \dots & \sigma_{ngbhd}(F(m^2),r) \end{bmatrix} \tag{10}$$

Now, using each entry in $\Sigma_{ngbhd}$, we can update Equation (4) to the following:

$$w_{ij} = exp\{- \parallel F(i) - F(j) \parallel_2^2 *(\sigma_{ngbhd}(F(i),r) \cdot \sigma_{ngbhd}(F(j),r))\} \\ * \begin{cases} e^{\frac{-\|X(i)-X(j)\|_2^2}{\sigma_X}} & \text{if } \parallel X(i) - X(j) \parallel_2 < R \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

$$for \quad i = 1, \dots, m^2$$

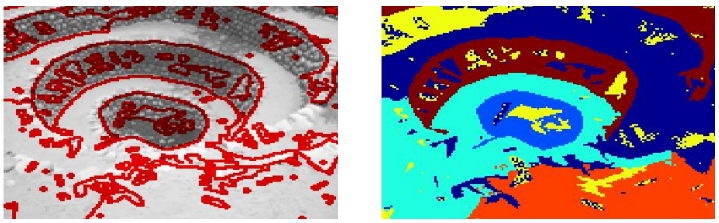
One advantage of tuning the features by the joint multiplication of the $\sigma_{ngbhd}(F(i),r) \cdot \sigma_{ngbhd}(F(j),r))$ pairs is to capture the correlation of the neighborhood features between pixel $i$ and pixel $j$ in constructing the weight edges. Note that in this updated Equation (11), the spatial component $\parallel X(i) - X(j) \parallel_2 < R$ is unchanged from previous Equation (4). We present other possible incorporation methods later in Section 4.

In our version, note that we are multiplying, instead of dividing, $- \parallel F(i) - F(j) \parallel_2^2$ by $\sigma_{ngbhd}(F(i),r) \cdot \sigma_{ngbhd}(F(j),r))$. This choice is simply due to the method of construction of our tuning parameter. For a given radius $r$, if the $\sigma_{ngbhd}(F(i),r)$ is low, the local neighborhood variation of the features around pixel $i$ will be low, implying that the pixels within the neighborhood share similar feature values, indicating group structure among the pixels. In the original NCut algorithm, Shi and Malik divided the features $- \parallel F(i) - F(j) \parallel_2^2$ by a small $\sigma_F$ value which is similar to us multiplying the features by a large $\sigma_{ngbhd}(F(i),r) \cdot \sigma_{ngbhd}(F(j),r))$ value.

Similarly, for a given neighborhood radius $r$ around pixel $i$ and pixel $j$, if the pixels within and around each neighborhood of pixels $i$ and $j$ are similar in terms of their local feature values, then $\sigma_{ngbhd}(F(i),r) \cdot \sigma_{ngbhd}(F(j),r))$ will be low, making $w_{ij} = exp[- \parallel F(i) - F(j) \parallel_2^2 *(\sigma_{ngbhd}(F(i),r) \cdot \sigma_{ngbhd}(F(j),r))]$ high. We want pixels that are similar to its neighbors to have strong weight connections in the affinity

matrix.

After incorporating $\Sigma_{ngbhd}(r)$ with $r = 3$ into the affinity matrix according to Equation (11), we run the NCut algorithm using the other pre-specified parameter values ($Segments = 6$, $\sigma_X = 0.3$, $R = 10$). The segmentation results for the well image is shown below in Figure 7:



(a) Flexible Segmentation (b) Flexible Segmentation (color)

Figure 7: (a)-(b) The NCut segmentation after incorporating $\Sigma_{ngbhd}(r = 3)$ with $Segments = 6$, $\sigma_X = 0.3$, $R = 10$

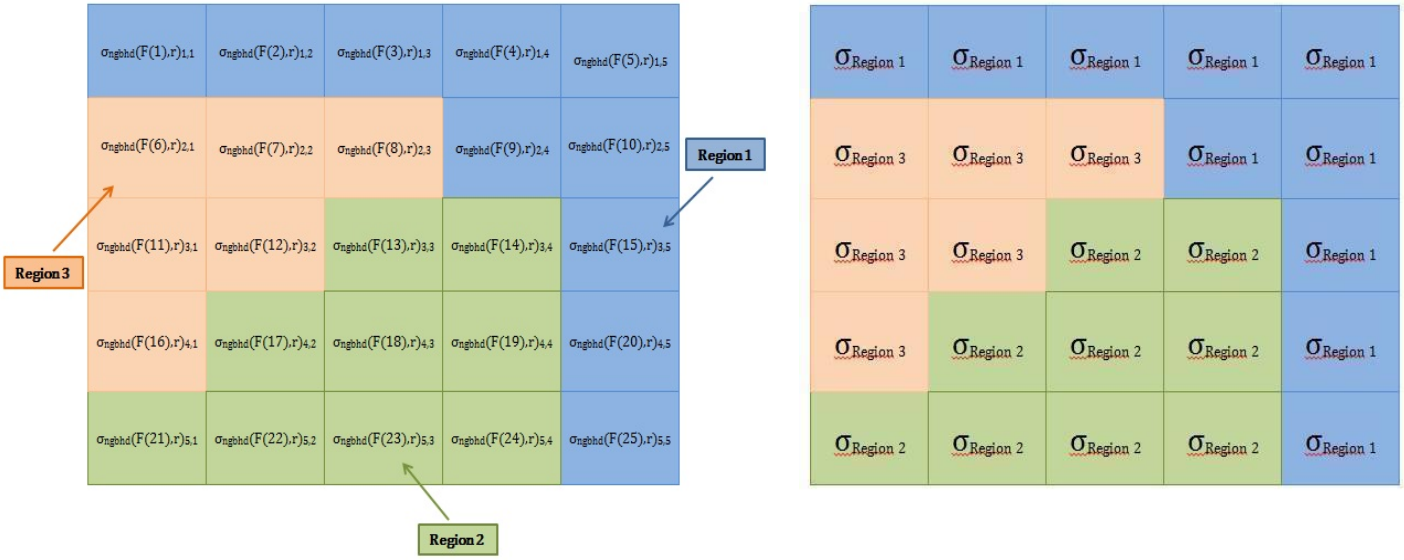
Note that a higher $r$ would give a more local segmentation and a lower $r$ would give a more global segmentation. We tried $r$ from 1 to 6 and we chose $r = 3$ as a middle ground for illustration purposes. From the result in Figure 7 above, we can see that the segmentation is patchy and more local than the original segmentation. From our perspective, we believe that this segmentation is too detailed. Instead, we would like to find a more global segmentation than Figure 7(a) but still be able to add flexibility to the construction of the affinity matrix. We believe since the $\Sigma_{ngbhd}(r)$ contains multiple different varying values $\sigma_{ngbhd}$ for most pixels, incorporating these values in computation of the weight edges apparently gives rough, non-uniform segmentation. Thus, to fix this, we seek to smooth out the varying values in the $\Sigma_{ngbhd}(r)$ matrix by first finding 'reasonable' regions in the image corresponding to the same regions in $\Sigma_{ngbhd}(r)$, and then for a specific region, find a single constant $\sigma_{ngbhd}$ measure that most appropriately represents the $\sigma_{ngbhd}$ values in that region.

### 3.3 Shape-Based Local-Variation Tuning Matrix, $\Sigma_{shaped}$

Our goal now is to construct a Shape-Based Local-Variation Tuning matrix, $\Sigma_{shaped}$. We find certain 'shaped regions' of interest in the image, and then find a representative $\sigma_{ngbhd}$ value from the corresponding region in the $\Sigma_{ngbhd}$ matrix. Finding these regions allow us to customize and tune the weight edges according to specific image

regions of interest.

To illustrate the concept of $\Sigma_{shaped}(r)$ more clearly, assume we have a digital image of size 5 x 5 and we have constructed a corresponding $\Sigma_{ngbhd}(r)$ of 5 x 5 from it:




(a) Selected Regions of $\Sigma_{ngbhd}(r)$ (b) The Final $\Sigma_{shaped-ngbhd}(r)$

Figure 8: The Final $\Sigma_{shaped-ngbhd}(r)$

Then, since the image and $\Sigma_{ngbhd}(r)$ share the same pixel locations, we select three 'shaped regions' in the image and the corresponding three regions in $\Sigma_{ngbhd}(r)$ as shown in Figure 8(a). To construct the Shape-Based Local-Variation Tuning Matrix, $\Sigma_{shaped}(r)$ shown in Figure 8(b), for each region, we find a representative aggregate measure of the $\sigma_{ngbhd}$ values and replace them by that aggregate measure, $\sigma_{Region}$. For example, for Region 1, we might take the mode of the $\sigma_{ngbhd}$ values and replace them with that mode value, $\sigma_{Region1}$. Mathematically,

$$\sigma_{Region1} = mode\ \ (\{\sigma_{ngbhd}(F(i), r) : \sigma_{ngbhd}(F(i), r) \in \text{Region 1}\}\ ) \tag{12}$$

So, how do we choose these 'shaped regions'? A natural way to pick these regions is to just use the segments from the original NCut algorithm. There are two advantages in doing this. One, instead of blindly picking random regions, the original NCut segments give a good starting segmentation for our procedure. Two, if the original

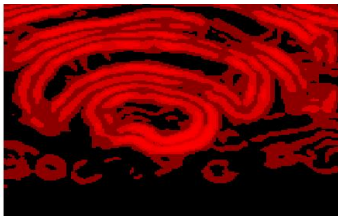
segments are somewhat reasonable, we are able to further improve and fine tune our affinity matrix at these regions to give a better and more accurate segmentation.

Then, how do we find a single *representative* aggregate measure, $\sigma_{Region}$ of the $\sigma_{ngbhd}$ values for a specific region? There are several possibilities. For a particular region, we could take the equal-weighted mean, exponentially-weighted mean, median, mode, equal-weighted variance, decaying-weighted variance, or a combination of any of the above of the $\sigma_{ngbhd}$ values to find $\sigma_{Region}$. We have experimented with all of the measures mentioned above, and found the **mode** measure works best. Among the aggregate measures mentioned, using the mode gave the least extreme local segmentation. Intuitively, the mode measure makes sense: since for a particular region, given that the $\sigma_{ngbhd}$ values are a function of the neighborhood around them, there will be some $\sigma_{ngbhd}$ values that include information from pixels outside its region; hence, we want the $\sigma_{ngbhd}$ value that occurs most frequently, which most likely represent the local variation of the features in that specific region. If there is a region with no repeating $\sigma_{ngbhd}$ values, the mode chooses the lowest $\sigma_{ngbhd}$ value. In this case, the lowest $\sigma_{ngbhd}$ value will give high affinities to the pixels in that region, indicating a group structure which is no other than the original NCut segment itself.

In summary, to construct our final $\Sigma_{shaped}(r)$ matrix, we first use the original NCut segments as the selected *regions* in the $\Sigma_{ngbhd}(r)$ matrix,and then second, for each region in the $\Sigma_{ngbhd}(r)$ matrix, we replace all the $\sigma_{ngbhd}$ values in that region with the mode value.

Shown in Figure 9 on the next page are heatmaps of the $\Sigma_{ngbhd}(r)$ matrix and the $\Sigma_{shaped}(r)$ matrix constructed using six original NCut segments, with neighborhood radius $r = 3$.

As expected, in Figure 9(b), the heatmap shows that the $\Sigma_{shaped}(r = 3)$ only has six unique values. The $\Sigma_{shaped}(r = 3)$ has uniform $\sigma_{region}$ values at only specific regions in the image. Compared to the heatmap of $\Sigma_{ngbhd}(r = 3)$ in Figure 9(a), $\Sigma_{shaped}(r = 3)$ has more uniform values than the values in $\Sigma_{ngbhd}(r = 3)$. By constructing the $\Sigma_{shaped}$ matrix, we have managed to smooth the varying $\sigma_{ngbhd}$ values as desired. Again, we do this to construct a more uniformly tuned affinity matrix to get a more desired global segmentation but also allow flexibility in the algorithm in



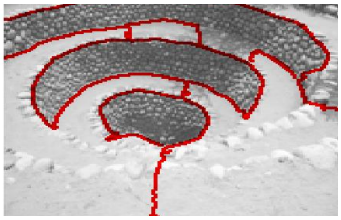
(a) Heatmap of $\Sigma_{ngbhd}(r = 3)$ matrix

(b) Heatmap of $\Sigma_{shaped}(r = 3)$ matrix with six regions

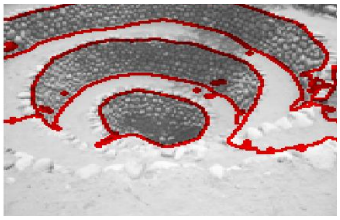
Figure 9: (a)-(b) Heatmaps of the $\Sigma_{ngbhd}(r = 3)$ and the $\Sigma_{shaped}(r = 3)$ matrix of the well image

the tuning of the edge weights. One possible weakness of this approach is that it is highly dependent on the original NCut segments. If the original NCut segments are really off and unstructured (for example, Figure 3(a)), the $\Sigma_{shaped}$ matrix will assume unstructured regions as well.

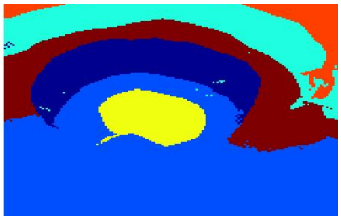
Using the well image, we construct our Shape-Based Local-Variation Tuning matrix of radius 3, $\Sigma_{shaped}(r = 3)$, and then incorporate it into the affinity matrix using Equation (11). The original segmentation and the segmentation using our methodology for the well image are both shown below:



(a) Original segmentation



(b) Shape-based flexible segmentation



(c) Shape-based flexible segmentation (color)

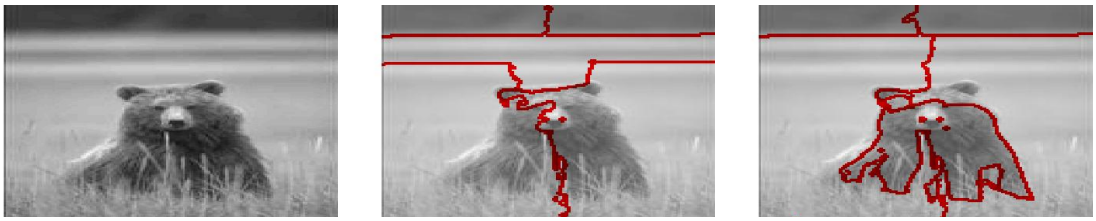
Figure 10: (a) The original NCut segmentation with $\sigma_F = 0.1$; (b)-(c) The NCut segmentation after incorporating $\Sigma_{shaped}(r = 3)$; shared parameters: $Segments = 6$, $\sigma_X = 0.3$ and $R = 10$.

From Figure 9, we can see that the segmentation result using our proposed methodology outperformed the original segmentation result in Figure 9(a). Recall from our discussion and using our definition of *reasonable* in Section 2.5, the original algorithm failed to give a reasonable segmentation of the well image. Clearly, comparing both Figures 9(a) and 9(b), our proposed method gave *more* reasonable segments. Our improved algorithm has successfully segmented out the 2nd level of the well image given the existence of the tree shadow. Note that our segmentation result is still

not entirely perfect. By inspection, there are still some levels of the well image that are not segmented correctly. For example, the 3rd level at the bottom of the image was not segmented out as a cluster by itself. However, our segmentation result is still an improvement from the original segmentation. Using our proposed methodology, we may be able to improve the original segmentation results of the well image. In the following subsection, we will look at some modifications to the construction of the $\Sigma_{shaped}$ matrix in hopes to further improve our segmentation result.

### 3.3.1 Some Modifications to The Construction of the $\Sigma_{shaped}$ Matrix

In this subsection, we will use a different image example, an image of a bear hiding behind tall grass. This image was taken from The Berkeley Segmentation Dataset and Benchmark at `http://www.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/`. Shown below are Figures of the original image, and the segmentation results from the original NCut algorithm and from using our proposed methodology:



(a) A bear behind tall grasses (b) Original segmentation (c) Shape-based flexible segmentation

Figure 11: (a) A bear behind tall grasses image example; (b) The original NCut segmentation; (c) The NCut segmentation after incorporating $\Sigma_{shaped}(r = 6)$; shared parameters: *Segments* = 5, $\sigma_X$ = 0.3 and $R = 10$.

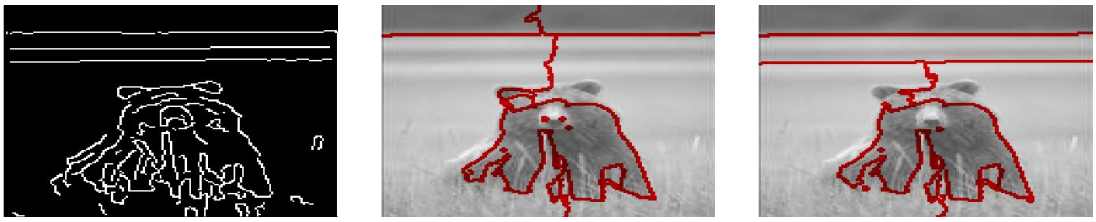
From the above Figure, we can see that the original NCut algorithm did not manage to segment out the bear from the tall grasses. On the other hand, our shape-based flexible NCut algorithm managed to segment out the bear from the tall grass. However, our algorithm fails to put the bear's right ear in the same segment as the bear.

**Canny Enhanced $\Sigma_{shaped}$ Matrix**

Here, we try to improve our segmentation result by using the Canny Edge Detector to enhance the construction of the $\Sigma_{shaped}$ matrix. The Canny Edge Detector is a algorithm that detects edges in an image using the gradient in the pixel features [8]. In general, an edge of a digital image is characterized by the sudden change in feature

values on each side of the edge. The Canny Edge method then detects edges by finding large gradients or changes in feature values between pixels. The result of running the standard Canny Edge Detector on the bear image can be seen in Figure 12(a). Using the Canny Edge Detector, we find the edges in the image, and then replace the values in $\Sigma_{shaped}$ with the values in $\Sigma_{ngbhd}$ corresponding to the Canny edges. Through this method, we incorporate the $\sigma_{ngbhd}$ values along the image edges into the $\Sigma_{shaped}$ matrix. This way, the $\sigma_{ngbhd}$ values along the image edges are represented in the $\Sigma_{shaped}$ matrix instead of just aggregate values. This would help the $\Sigma_{shaped}$ better identify edges in the image.

Below are the segmentation results:



(a) Canny edges

(b) Shape-based flexible segmentation

(c) Canny enhanced shape-based flexible segmentation

Figure 12: (a) The edges generated from the Canny Edge Detector; (b)-(c) The NCut segmentations after incorporating $\Sigma_{shaped}(r = 6)$ and Canny Enhanced $\Sigma_{shaped}(r = 6)$, respectively, with *Segments* $= 5$, $\sigma_X = 0.3$ and $R = 10$.

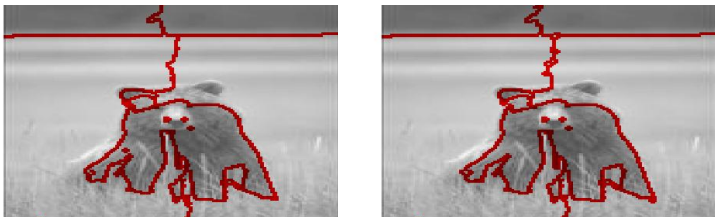
From Figure 12 above, we can see that this Canny enhanced method did not help improve our segmentation. In this case, the Canny enhanced method makes the segmentation result worse. This new method does not only fail at grouping the right ear of the bear, it also lost the left ear of the bear in the segmentation. It is not surprising to see that the Canny enhanced method gave similar results to the shape-based flexible NCut segmentation. The Canny enhanced method only introduced a relatively small number of $\sigma_{ngbhd}$ values into the $\Sigma_{shaped}$ matrix, while most of the values in $\Sigma_{shaped}$ matrix are still dominated by the $\sigma_{Region}$ values. Thus, the relatively small number of $\sigma_{ngbhd}$ values in the $\Sigma_{shaped}$ matrix did not really affect the segmentation.

**Bounding Box Method**

Here, we try to improve our segmentation by defining a bounding box around each of our original NCut segments and hence extending the bounds used to compute the

$\sigma_{Region}$ values. The motivation behind the use of the box was to incorporate more local information in the shape-based flexible NCut segmentation. By doing this, we include a larger region when computing the $\sigma_{Region}$ mode value for a specific region to capture possible missing variations surrounding the edges that region. We hoped that including more neighboring $\sigma_{ngbhd}$ values might better inform the distribution of the $\sigma_{ngbhd}$ values of a specific region and obtain a more *representative* mode value .

Below are the segmentation results:



(a) Shape-based flexible segmentation

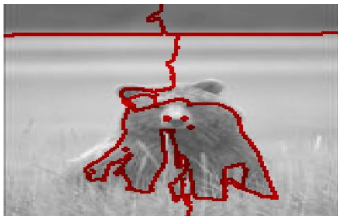
(b) Bounding box enhanced shape-based flexible segmentation

Figure 13: (a)-(b) The NCut segmentations after incorporating $\Sigma_{shaped}(r = 6)$ and the Bounding Box Enhanced $\Sigma_{shaped}(r = 6)$, respectively, with $Segments = 5$, $\sigma_X = 0.3$ and $R = 10$.

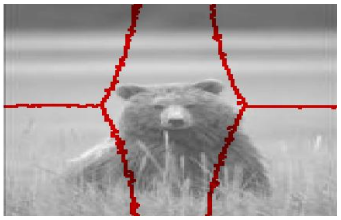
From Figure 13, we can see that this bounding box bound method did not help improve our segmentation result at all. This bounding box bound method resulted in the same original segmentation in 13(a). The bounding box method which extends the size of the original NCut segments for the computation of the $\sigma_{Region}$ values did not change the segmentation. Given that a original NCut segment was already large, adding a bounding box around it only includes a relatively small number of extra $\sigma_{ngbhd}$ values to the larger boxed region, and taking the mode of the larger boxed region to compute $\sigma_{Region}$ will most likely result in the same mode value of the original NCut segment itself in the $\Sigma_{shaped}$ matrix since the majority of the $\sigma_{ngbhd}$ values are in the smaller original NCut segment.

**Exponentially Weighted Variance Measure for $\sigma_{Region}$ Values**

Here, we try to improve our segmentation result by exploring the use the exponentially weighted variance as the aggregate measure of $\sigma_{Region}$ values instead of using the mode measure. We would like to look at the variation of the $\sigma_{ngbhd}$ values themselves. The segmentation result is shown in Figure 14 on the next page.



(a) Shape-based flexible segmentation



(b) Exponentially weighted variance shape-based flexible segmentation

Figure 14: (a)-(b) The NCut segmentations after incorporating $\Sigma_{shaped}(r = 6)$ and after using exponentially weighted variance to compute $\sigma_{Region}$ values, respectively, with $Segments = 5$, $\sigma_X = 0.3$ and $R = 10$.

From Figure 14, we can see that this exponentially weighted variance method did not help improve our segmentation result. Taking the variance of a variance measure resulted in a smaller value. Hence, the decaying variance method gave really small $\sigma_{Region}$ values, resulting in too global and unstructured segments (compare to Figure 3(a)). This decaying variance method performs the worst among them all.

In conclusion, none of our modified methods helped at further improving our NCut segmentations. So far, our shape-based flexible NCut algorithm is still the better option.

# 4 Incorporating $\Sigma_{shaped}$ into the Affinity Matrix

Recall from section 3.2, we explored one possibility of incorporating $\Sigma_{shaped}$ into the affinity matrix using Equation (11). Recall that the advantage of tuning the features by the join multiplication of the $\sigma_{ngbhd}(F(i), r) \cdot \sigma_{ngbhd}(F(j), r))$ pairs in Equation (11) is to capture the correlation of the neighborhood features between pixel $i$ and pixel $j$ in constructing the weight edges. In this section, we explore two additional ways to incorporate $\Sigma_{shaped}$ values into the affinity matrix through incorporating the $\sigma_{Region}$ values into computing the edge weights.

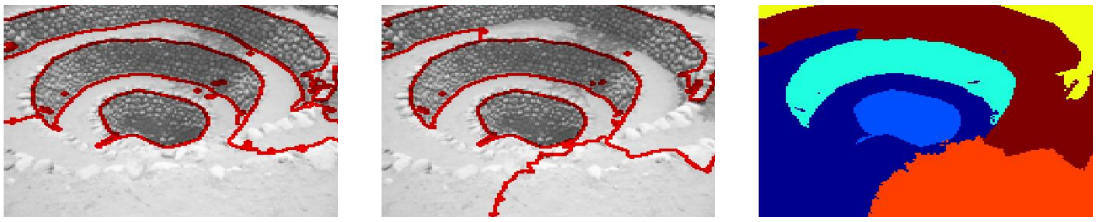
## 4.1 Method 1: Tuning Each Individual Feature

In this approach, we individually tune each pixel's features by its corresponding $\sigma_{ngbhd}$ value. For example, for local features $F(i)$, $F(j)$, we individually multiply $F(i)$ by

$\sigma_{ngbhd}(F(i), r)$ and individually multiply $F(j)$ by $\sigma_{ngbhd}(F(j), r)$. That is,

$$w_{ij} = exp\{- \parallel F(i) \cdot \sigma_{ngbhd}(F(i), r) - F(j) \cdot \sigma_{ngbhd}(F(j), r) \parallel_2^2\} * \begin{cases} e^{\frac{-\|X(i)-X(j)\|_2^2}{\sigma_X}} & \text{if } \parallel X(i) - X(j) \parallel_2 < R \\ 0 & \text{otherwise} \end{cases} \tag{13}$$

Shown below is the segmentation result of the well image using our $\Sigma_{shaped}$ methodology but incorporated the affinity matrix using Equation (13) instead:



(a) Shape-based flexible segmentation

(b) Shape-based flexible segmentation with Method 1

(c) Shape-based flexible segmentation with Method 1 (color)

Figure 15: (a) The shape-based flexible NCut segmentation; (b)-(c) The NCut segmentations after incorporating $\Sigma_{shaped}(r = 3)$ using Equation (13); shared parameters: $Segments = 6$, $\sigma_X = 0.3$ and $R = 10$.

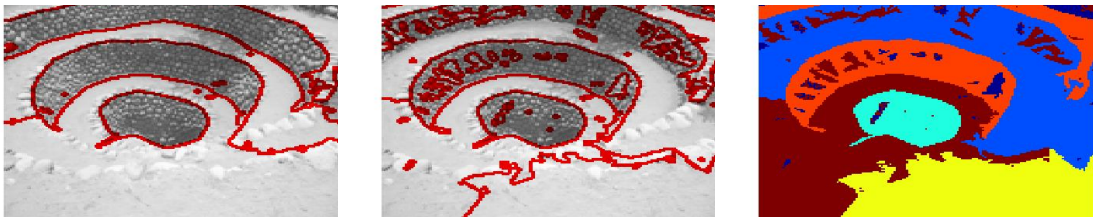
As one can see from Figure 15(b) above, the shape-based flexible NCut algorithm using Method 1 did not manage to give reasonable segments compared to the shape-based flexible NCut segmentation in Figure 15(a). Using Method 1, the shape-based flexible NCut algorithm only managed to partially segment out the 2nd level of the well. Furthermore, similar to the original NCut segmentation in Figure 10(a), the shape-based flexible NCut algorithm using Method 1 did not manage to segment out the 3rd level of the well at the bottom of the image and "broke" the 3rd level up.

## 4.2 Method 2: Tuning Joint Features via Joint Addition

In this method, compared to Equation (11), instead of multiplying, we looked at adding the $\sigma_{ngbhd}(F(i), r)$ and $\sigma_{ngbhd}(F(j), r)$ pairs and then scale $- \parallel F(i) - F(j) \parallel_2^2$. This joint addition effect allow us to take into account the total independent local variation of pixels $i$ and $j$. That is,

$$w_{ij} = exp\{- \parallel F(i) - F(j) \parallel_2^2 * (\sigma_{ngbhd}(F(i), r) + \sigma_{ngbhd}(F(j), r))\} * \begin{cases} e^{\frac{-\|X(i)-X(j)\|_2^2}{\sigma_X}} & \text{if } \parallel X(i) - X(j) \parallel_2 < R \\ 0 & \text{otherwise} \end{cases} \quad (14)$$

Shown below is the segmentation result of the well image using our $\Sigma_{shaped}$ methodology but incorporated the affinity matrix using Equation (14) instead:



(a) Shape-based flexible segmentation

(b) Shape-based flexible segmentation with Method 2

(c) Shape-based flexible segmentation with Method 2 (color)

Figure 16: (a) The original NCut segmentation; (b)-(c) The NCut segmentations after incorporating $\Sigma_{shaped}(r = 3)$ using Equation (14); shared parameters: *Segments* = 6, $\sigma_F = 0.1$, $\sigma_X = 0.3$ and $R = 10$.

As one can see from Figure 16 above, using Method 2, the shape-based flexible NCut algorithm gives a more 'patchy' segmentation compared to the original NCut segmentation. Since Method 2 aims to capture the total local variation between independent pixels $i$ and $j$, we would expect to see a high joint additional tuning effect on the features, resulting in a more local segmentation. As mentioned before, comparing to Figure 16(b), we are looking for a more global segmentation but incorporate sufficient flexibility and local variation to the segmentation. Furthermore, similar to the original NCut segmentation in Figure 16(a), the shape-based flexible NCut algorithm using Method 2 in Figure 16(b) did not manage to segment out the 3rd level of the well at the bottom of the image and "broke" the 3rd level up.

In summary, comparing Figure 16(b), 15(b) and 10(b), using Equation (11) on the shape-based flexible NCut algorithm gives the most reasonable segmentation. Thus, for our choice of approach, we use Equation (11) in our shape-based flexible NCut algorithm. In this paper, the phrase "shape-based flexible NCut algorithm" will now refer to using Equation (11) to incorporate $\Sigma_{shaped}$ into the NCut algorithm.

# 5 Evaluating Segmentation Results

In this section, we will try to evaluate our shape-based flexible segmentation result using existing evaluation measures. We also use these measure to find an appropriate $r$ parameter for our $\Sigma_{ngbhd}(r)$ matrix values that gives the "best" segmentation. We explored three evaluation measure for our segmentation results: the CH Index [10], the Silhouette Measure [11], and the Gap Statistic [12].

## 5.1 CH Index

The CH Index is an intuitive evaluation measure that uses the ratio of the Between Cluster Sum of Squares (BCSS) and the Within Cluster Sum of Squares (WCSS) of the segmentation. The BCSS measures how segments are different among each other. A high BCSS value indicates that individual segments are very different from each other, what we expect from a good segmentation. The WCSS measures how different the pixels are within each segment. A low WCSS value indicates that pixels within their segments are very similar, again is what we expect from a good segmentation. In particular, WCSS, BCSS and ultimately the CH Index are defined as follows:

Recall,
$$\bar{F}_{grand} = \text{grand mean of features} = \frac{1}{n}\sum_{i=1}^{n} F(i)$$

Then,
$$BCSS(K) = \sum_{j=1}^{K} \sum_{F(i)\in G_j} |G_j| \cdot \parallel F(i) - \bar{F}_{grand} \parallel_2^2 \tag{15}$$

$$WCSS(K) = \sum_{j=1}^{K} \sum_{F(i)\in G_j} \parallel F(i) - \bar{F}_j \parallel_2^2 \tag{16}$$

And,

$$CH(K) = \frac{BCSS(K)/(K-1)}{WCSS(K)/(n-K)}$$

$$\text{where,} \quad \begin{aligned} K &= \text{number of segments,} \\ n &= \text{total number of pixels and,} \\ G_j &= \text{segment j} \end{aligned} \tag{17}$$

Generally, a "good" segmentation will have a high BCSS value and a low WCSS value. Thus, we choose $K$, the number of segments, that gives the **highest** CH index.

## 5.2 Average Silhouette Measure

The average silhouette measure, $Silhouette_{avg}$ is a measure of how closely grouped pixels are within their own segments and how loosely these pixels might be matched on average to pixels from neighboring segments. A silhouette measure close to 1 implies that the pixels are, on average, in their appropriate segments, while a silhouette close to -1 implies that the pixels, on average, have been assigned to the wrong segments. In particular, the average silhouette measure is defined as follows:

$$Silhouette_{avg}(K) = \frac{1}{K}\sum_{j=1}^{K}\frac{1}{n_j}\sum_{i=1}^{n} S_i \qquad (18)$$

$$\text{where,} \qquad S_i = \frac{b(i) - a(i)}{max[a(i), b(i)]}$$

$a(i)$ = the average squared Euclidean distance from the $i^{th}$ pixel to the other pixels in the same segment as $i$, and

$b(i)$ = the minimum average distance from the $i^{th}$ pixel to pixels in a different segment, minimized over all segments

## 5.3 Gap Statistic

The Gap statistic is based on the following idea. The Gap statistic compares the observed Within Cluster Sum of Squares, WCSS, to the expected Within Cluster Sum of Squares ,$WCSS_{unif}$, if we instead had uniformly distributed pixels over the segment containing the pixel. Then, the Gap statistic for $K$ segments is defined as follows:

$$Gap(K) = log\ WCSS(K) - log\ WCSS_{unif}(K) \qquad (19)$$

The quantity $log\ WCSS_{unif}(K)$ is computed by simulation: it is the average log WCSS(K) over some simulated uniform pixels. Thus, the higher the Gap statistic is, the better the segmentation result.

## 5.4 Evaluating Segmentations of Well Image

In Figure 17, we show segmentation results using our proposed methodology of the well images for varying number segments, $K$, and neighborhood radii, $r$, used in computing the $\Sigma_{shaped}(r)$. The *purple*, *green* and *red* boxes each respectively indicates the highest CH index, $Silhouette_{avg}$ and Gap statistic values. In Figure 18, we include the distributions of the relatively scaled values of the CH index, Silhouette average and Gap statistic measure for each of the segmentations. The CH index, Silhouette average, and Gap statistic are each scaled to a range between 0 and 1 so we can relatively compare the measures.

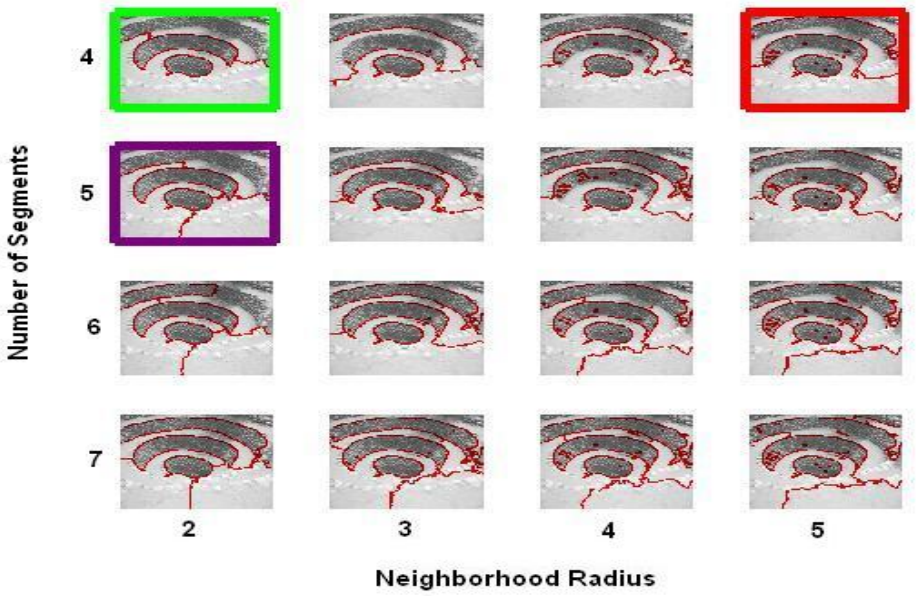



Figure 17: Shape-based flexible segmentations for *Segments* from 4 to 7 and Neighboorhood Radius from 2 to 5; The *purple*, *green* and *red* boxes respectively indicates the highest CH index, $Silhouette_{avg}$ and Gap statistic values.

From Figure 17, we can see that each of the different evaluation measures has selected its "best" segmentation. Note that the "best" segmentation result is different for each evaluation measure. The conclusion across the evaluation measures is inconsistent. However, looking at Figure 18, the scaled values of the CH index, the Silhouette average, and the Gap statistic for the segmentation with 5 segments and neighorhood radius 1 are similarly high and close to 1. The Silhouette average and Gap statistic both selected the segmentations with 4 number of segments and

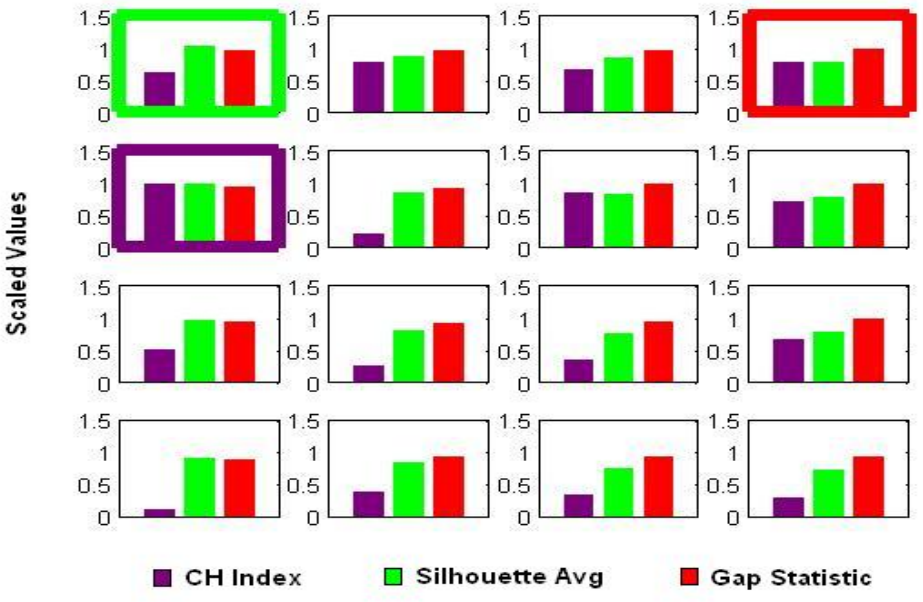



Figure 18: The distributions of the relatively scaled values of CH index, Silhouette Average and Gap Statistic measure for each corresponding segmentations in Figure 17.

respectively, neighborhood radius of 2 and 5 to be their "best" segmentations; but looking at Figure 18, these "best values" are just slightly higher than the values in the segmentation with 5 segments and neighborhood radius 1. By looking at the distribution of the scaled values of the evaluation measures, we can say that the evaluation measures are in close agreement of the segmentation with 5 segments and radius 1 being the best, and they appear to be just slightly inconsistent.

By inspection, none of the evaluation measure selected the "best" segmentation of the well image, which is the segmentation with *five* number of segments and a neighborhood radius of *three*. Like any other clustering problems, image segmentation runs into the problem of being able to choose an appropriate number of segments. One could use cross validation to select the number of segments. Nevertheless, in unsupervised learning problems like image segmentation, evaluating the segmentations it's a hard and open problem. Unfortunately, none of our proposed evaluation measure seems to find the "best" segmentation by inspection.

# 6 Experiments

In this section, we experiment with our shape-based flexible NCut algorithm on other image examples. The segmentations are presented in Figure 19 on the next page.

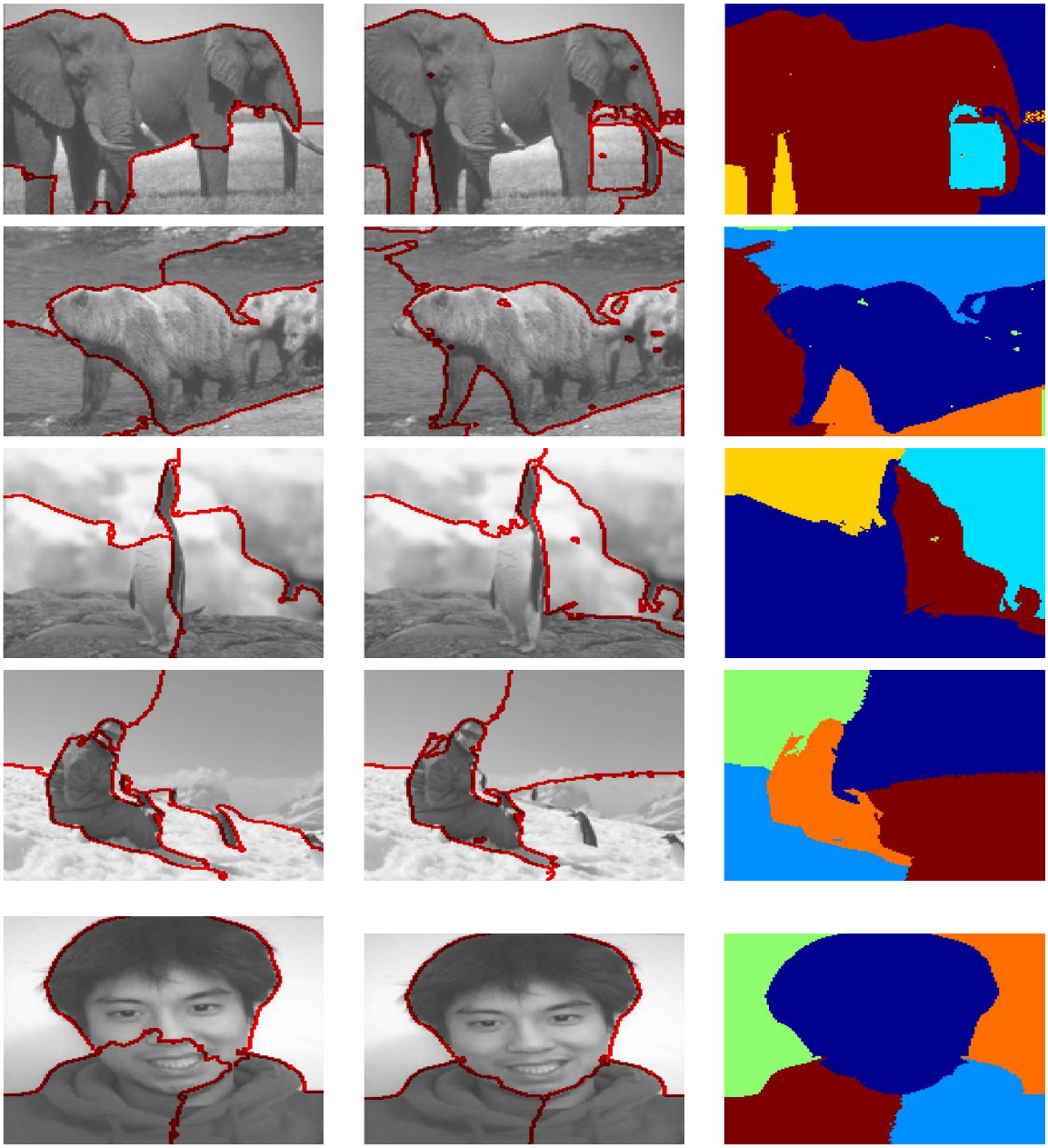
For all of our example images, by inspection, the shape-based flexible NCut algorithm gives better segmentations compared to their respective original NCut segmentations. For example, for the elephants, the shape-based flexible NCut algorithm managed to grab the legs of both the elephants as well as the trunk of the elephant at the left, both of which the original NCut algorithm failed to do. Furthermore, if we look at the image of the mother bears and cubs, the shape-based improved NCut algorithm managed to capture the front leg of the mother bear, while the original NCut algorithm did not. Moreover, for the image of the penguin, the shape-based flexible NCut algorithm succeeds in segmenting out the whole penguin from the background when the original NCut algorithm only identifies the black fur that heavily contrasts with the white background from the penguin. In addition, for the image of Professor Nugent, the shape-based flexible NCut algorithm separates Professor Nugent from the background while the original NCut algorithm does slightly worse. Finally, if we look at the face segmentation of Yi Xiang Chong, the flexible shape-based NCut algorithm segments the entire face, while the original NCut algorithm divides the face.

From the segmentation results in Figure 19, we have successfully managed to incorporate local variation and flexibility into the NCut algorithm.

# 7 Robustness

In this section, we will look at the robustness of both the original NCut algorithm and our proposed shape-based flexible NCut algorithm in the presence of random white noise. The motivation behind doing this is to look at possibly applying our flexible shape-based NCut algorithm on old, dirty digitally scanned pictures or images taken by row resolution cameras.

We test both algorithms on two noisy images of the spiraling well. In the first well image, we add normal random noise with mean 0 and variance 20, $\epsilon_1 \sim N(0, 20)$, to all pixels in the image. In the second well image, we add normal random noise with mean 0 and variance 60, $\epsilon_2 \sim N(0, 60)$, to all pixels. The segmentation results



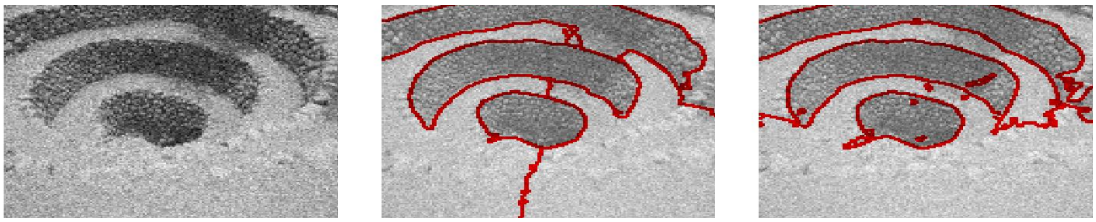
(a) Original segmentation

(b) Shape-based flexible segmentation

(c) Shape-based flexible segmentation (color)

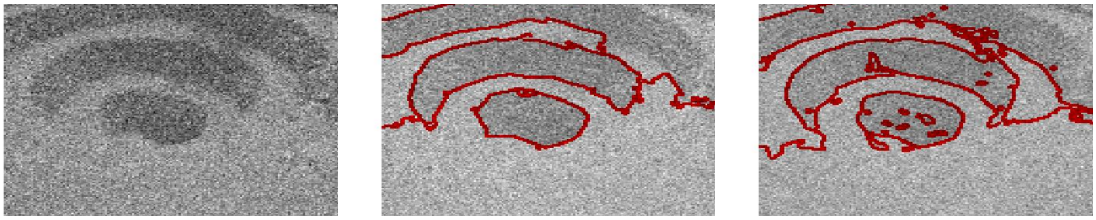
Figure 19: (a)-(c) Original and shape-based flexible NCut segmentations of example images: elephants (*Segments* $= 4$, $r = 4$), a mother bear with cubs (*Segments* $= 5$, $r = 10$), a penguin (*Segments* $= 4$, $r = 6$), Professor Nugent (*Segments* $= 4$, $r = 1$), and Yi Xiang Chong (*Segments* $= 5$, $r = 1$); shared parameters: $\sigma_X = 0.3$, $R = 10$.

of both these algorithms for these two noisy images are shown in Figure 20 and 21:



(a) Noisy well image 1 (b) Original segmentation (c) Shape-based flexible segmentation

Figure 20: (a) The spiraling well image with $\epsilon_1 \sim N(0, 20)$ added to all pixels; (b) The original NCut segmentation of the noisy well image 1; (c) The shape-based flexible segmentation of the noisy well image 1 with neighorhood radius, $r = 3$; shared parameters: $\sigma_X = 0.3$, $R = 10$.



(a) Noisy well image 2 (b) Original segmentation (c) Shape-based flexible segmentation

Figure 21: (a) The spiraling well image with $\epsilon_2 \sim N(0, 60)$ added to all pixels; (b) The original NCut segmentation of the noisy well image 2; (c) The shape-based flexible segmentation of the noisy well image 2 with neighorhood radius, $r = 3$; shared parameters: $\sigma_X = 0.3$, $R = 10$.

From Figure 20 and Figure 21 above, we can see that the NCut algorithm itself (the original and which the shape-based flexible NCut algorithm is based on) is quite robust towards noisy images in general. In Figures 20(b), 20(c), 21(b), and 20(c), the NCut algorithm still manages to find segments that correspond to the levels of the well in both of the noisy images.

For a less noisy image like Figure 20(a), the original NCut algorithm and the shape-based NCut algorithm both perform well, and gave the same segmentations as before (compare with Figure 10(a) and 10(b)). For a noisier image like Figure 21(a), the shape-based flexible NCut algorithm seems to perform slightly better than the original NCut algorithm. In Figure 21(c), the shape-based flexible NCut algorithm seems to give more complete segments of the well levels. In particular, the

shape-based flexible NCut algorithm gave a complete 2nd level segment of the well image while in Figure 21(b), the original NCut algorithm gave a partially complete 2nd level segment of the well image. Furthermore, in Figure 21(c), the shape-based flexible NCut algorithm manages to segment out some of the tiny little rocks in the middle. When faced with a noisy image, it may be better to have a slightly more local segmentation than a global segmentation so that we can obtain more information about the variation in the noisy image.

In summary, both the original NCut algorithm and the shape-based flexible NCut algorithm are robust with slightly noisy images. However, the shape-based flexible NCut algorithm may perform slightly better for noisier images.

# 8 Future Work and Discussion

Our proposed shape-based flexible NCut algorithm seems to perform moderately better and to be potentially more robust than the original NCut Algorithm. However, we believe there is more work that could be done to further improve the segmentation results of the NCut Algorithm. So far, we have just redefining the $\sigma_F$ tuning parameter to improve the segmentations. One possible future approach would be to additionally redefine the spatial tuning parameter, $\sigma_X$. For example, one could make $\sigma_X$ a function of the image features. In this research paper, we have only looked at black and white images. examining the performance on color or textured images. The proposed shape-based flexible methodology is still applicable to images with multiple feature vectors (for example, RGB content).

One might argue that we could use cross validation to select the $\sigma_F$ value instead of calculating individual neighborhood tuning values for each pixel. However, one challenge of using cross validation to select the tuning parameters is picking an appropriate evaluation measure for "good" segmentations. In unsupervised learning like image segmentation, evaluating the segmentations is still a hard and open problem. Our method instead provides an intuitive framework to select the appropriate tuning parameter $\sigma_F$ based on just the variation of the features in the image itself. Furthermore, even if we could use cross validation to choose $\sigma_F$, we would still only have one constant $\sigma_F$ tuning parameter for the whole image instead of locally varying $\sigma_{Region}$ parameter values, which allows us to add flexibility to the algorithm.

One weakness of our proposed method is that the $\sigma_{ngbhd}(r)$ values are heavily dependent on the neighborhood radius, $r$, parameter. In a way, we come back to the original problem of trying to choose an appropriate value for an arbitrary parameter; in this case, the $r$ parameter. Nevertheless, the possible range of choosing $r$ is smaller than $\sigma_F$ since the largest possible neighborhood radius $r$ value it can take is half of the largest width or height of the image - the radius of the image itself. On the other hand, $\sigma_F$ can take any positive real number. One useful advantage of having the tuning parameter $\sigma_{ngbhd}(F(i), r)$ as a function of a $r$-sized neighborhood around each pixel $i$ is that the tuning parameter associated with pixel $i$ within a segmentation can always 'stretch' to reach out to capture the variation of the features from other regions of the image. This extra flexibility is extremely useful if the segmented object is part of another region not belonging to its segmentation. For example, for the penguin image in Figure 18, one can see that the original NCut algorithm failed to segment the whole penguin but only managed to grab its black fur since the black fur contrasted strongly with the white background. However, with neighborhood flexibility, the shape-based flexible NCut allowed the pixels in the original segmented black fur region to 'stretch out' to capture the variation of the white fur of the penguin, making it a single penguin segment.

In addition, having to compute our $\Sigma_{shaped}$ matrix adds additional computing time to the NCut algorithm. The original NCut algorithm takes, on average, 30 seconds to complete segmenting an image of size 160 x 160. With our proposed methodology, the NCut algorithm takes, on average, 2 minutes to complete segmenting an image, four times the amount of time taken by the original algorithm. Since we are using the original NCut segmentations to compute our $\Sigma_{shaped}$ matrix, we are in effect running the NCut algorithm twice. Furthermore, the computational time increases when the neighborhood radius $r$ increases. The original NCut algorithm itself becomes intractable when the image size is large. As shown in the original NCut paper, minimizing the NCut criterion is NP-complete. Approximations to our methodology will need to be developed.

We have introduced a more flexible approach in constructing the affinity matrix that may improve the segmentation and robustness of the original NCut Algorithm. However, there are still possible computational issues with our proposed methodology.

Future work should address speeding up the $\Sigma_{shaped}$ matrix construction. One could look at using heuristic methods , which are faster, to come up with approximate regions in the image for the construction $\Sigma_{shaped}$. Hopefully, our research will be helpful to the image segmentation field.

# References


[1] J.Shi and J.Malik, "Normalized Cuts and Image Segmentation, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 8, August 2000.

[2] K.S. Fu and J.K. Mui, "A Survey on Image Segmentation", *Pattern Recognition*, vol. 13, pp. 3-16, June 1980.

[3] D.L. Pham, C. Xu, and J.L. Prince, "Current Methods in Medical Image Segmentation", *Annual Review Biomedical Engineering*, vol. 2, pp. 315-317, 2000.

[4] T. Zuva, O.O. Olugbara, S.O. Ojo, and S.M. Ngwira, "Image Segmentation, Available Techniques, Development and Open Issues", *Canadian Journal on Image Processing and Computer Vision*, vol .2, no. 3, March 2011.

[5] F.R.K Chung, *Spectral Graph Theory*, American Mathematical Society, 1997.

[6] M. Meilă, and J. Shi, "A Random Walks View of Spectral Segmentation", *Proceedings of International Workshop on AI and Statistics (AISTATS)*, 2001.

[7] G.Welch and G.Bishop, "An Introduction to Kalman Filter", *Annual Conference on Computer Graphics and Interactive Techniques*, 2001.

[8] B.Green, "Canny Edge Detection Tutorial", *Drexel Autonomous Systems Lab*, 2004.

[9] L.Z. Manor, and P. Perona, "Self Tuning Clustering", *Neural Information Processing Systems (NIPS)*, 2004.

[10] T. Caliski, and J. Harabasz, "A Dendrite Method for Cluster Analysis", *Communications in Statistics*, vol. 3, pp. 1-27, 1974.

[11] P.J. Rousseeuw, "Silhouettes: a Graphical Aid to the Interpretation and Validation of Cluster Analysis", *Computational and Applied Mathematics*, vol. 20, pp. 53-65, 1987.

[12] R. Tibishirani, G. Walther, T. Hastie, "Estimating the number of clusters in a dataset via the gap statistic", *Journal of the Royal Statistical Society*, vol. 63, no. 2, pp. 411423, 2001.

[13] M. Meilă, and J. Shi, "Learning Segmentation by Random Walks", *Advances in Neural Information Processing Systems*,

[14] P.F. Felezenswalb, and D.P. Huttenlocher, "Efficient Graph-Based Segmentation", *International Journal of Computer Vision*, vol. 59, no.2, pp. 167-181, 2004.

[15] P. Kontscheider, M. Donoser, and H. Bischof, "Improving Affinty Matrices by Modified Mutual kNN-Graphs", *33rd Workshop of the Austrian Association for Pattern Recognition (AAPR/OAGM)*, May 2009.

[16] J. de la Porte, B.M. Herbst, W. Hereman, and S.J. van der Walt, "An Introduction to Diffusion Maps", *Applied Mathematics Division, Department of Mathematical Sciences, University of Stellenbosch, South Africa and Colorado School of Mines, United States of America*, 2008.

[17] Q.B. Xi, "An Improved Segmentation Algorithm Based on Normalized Cut", *2010 2nd International Conference on Computer Engineering and Technology*, vol. 7, 2010.

[18] C. Shalizi, "Nonlinear Dimensionality Reduction II: Diffusion Maps", *Data Mining Lecture Notes, Department of Statistics, Carnegie Mellon University*, 2009.

## Code Appendix

```
==================
1. createTable.m
==================
    % createTable: Creates and display segmentation results
    %
    %   createTable outputs a table of the NCut segmentation results
    %   with Number of Segments on the y-axis and Neighborhood Radius
    %   on the x-axis according to the specified parameters.

function createTable;

%% Default values of
% nbSegments = 5;     (default)
% sampleRadius = 10;  (default)
% sampleRate = 0.3;   (default)
% edgeVariance = 0.1; (default)

nbSegments = 5;
sampleRadius = 10;
sampleRate = 0.3;
edgeVariance = 0.1;

dataW.sampleRadius = sampleRadius;
dataW.sample_rate = sampleRate;
dataW.edgeVariance = edgeVariance;

sigmamatParam.YES    = 1; % 1 = yes sigmamat; 0 = no sigmamat
sigmamatParam.CANNY  = 0; % 1 = yes sigmamat with Canny; 0 = no sigmamat with Canny
sigmamatParam.RADIUS = 1; % neighborhood radius when computing sigmamat

%% Creating plots with "Number of Segments" vs "Sigmamat Radius"

% Image File
imgFile = 'spiral.jpg';

numSegs = [4 7]; sigmamatRad = [2 5];
[CHtest, Stest, Gtest, segObj_array, colObj_array] = ...
    createResults(numSegs, sigmamatRad, dataW, sigmamatParam, imgFile);
```

```
% Setting variables
nrow = numSegs(2)-numSegs(1)+1;
ncol = sigmamatRad(2)-sigmamatRad(1)+1;

% Find the index with the Silhouette measure closest to 1
Stemp = zeros(nrow, ncol);
for i = 1:nrow
    for j = 1:ncol
      nsize = size(Stest{i,j},2);
      temp = zeros(1,nsize);
      for k = 1:nsize
          temp(k) = mean(Stest{i,j}{k});
      end
      Stemp(i,j) = mean(temp);
    end
end
Stemp2 = Stemp+1; Stemp2 = Stemp2';
Sindex = find(Stemp2==max(Stemp2(:)));

% Find the index with the greatest CH measure
CHtest2 = CHtest ./ max(CHtest)';
CHindex = find(CHtest2==max(CHtest2(:)));

% Find the index with the greatest G measure
Gtest2 = Gtest ./ max(Gtest)';
Gindex = find(Gtest2==max(Gtest2(:)));

% Plot the colored segmentation graphs
counter = 1;

for i = 1:nrow
    for j = 1:ncol
            subplot(nrow,ncol,counter);
            imagesc(colObj_array{i,j}); axis off;
            counter = counter + 1;
    end
end

% Full screen segmentations
h = gcf;
% Maximize(h);
```

```
% Locate the figure(s) with the best CH measures
% CH index -> 'purple box'
hf = subplot(nrow, ncol, CHindex);
hf = get(hf, 'Position');
annotation('rectangle', hf, 'EdgeColor', [0.5,0,0.5], 'LineWidth', 5.0);

% Locate the figure(s) with the best G measure
% G index -> 'purple box'
hf = subplot(nrow, ncol, Gindex);
hf = get(hf, 'Position');
annotation('rectangle', hf, 'EdgeColor', 'Red', 'LineWidth', 5.0);

% Silhouette index -> 'green box'
hs = subplot(nrow, ncol, Sindex);
hs = get(hs, 'Position');
annotation('rectangle', hs, 'EdgeColor', 'g', 'LineWidth', 5.0);

% Plot the segmentation graphs
counter = 1;
figure;

for i = 1:nrow
    for j = 1:ncol
            subplot(nrow,ncol,counter);
            imagesc(segObj_array{i,j}); axis off;
            counter = counter + 1;
    end
end

% Full screen segmentations
h = gcf;
% Maximize(h);

% Locate the figure(s) with the best CH and Silhouette measures
% CH index -> 'purple box'
hf = subplot(nrow, ncol, CHindex);
hf = get(hf, 'Position');
annotation('rectangle', hf, 'EdgeColor', [0.5,0,0.5], 'LineWidth', 5.0);

% Locate the figure(s) with the best G measure
% G index -> 'purple box'
hf = subplot(nrow, ncol, Gindex);
```

```
hf = get(hf, 'Position');
annotation('rectangle', hf, 'EdgeColor', 'Red', 'LineWidth', 5.0);

% Silhouette index -> 'green box'
hs = subplot(nrow, ncol, Sindex);
hs = get(hs, 'Position');
annotation('rectangle', hs, 'EdgeColor', 'g', 'LineWidth', 5.0);

%% Distribution plots of the evaluation measures
counter = 1;
figure;

for i = 1:nrow
    for j = 1:ncol
            subplot(nrow,ncol,counter);
            v = [CHtest2(i,j) Stemp2(i,j) Gtest2(i,j)]';
            bh = bar(v, 'EdgeColor', [1 1 1]);
            set(gca, 'XTickLabel', {'CH Index', 'Silhouette Avg + 1', 'Gap Statistic'});
            ch = get(bh,'children');
            cd = [0.5 0 0.5; 0 1 0; 1 0 0];
            set(ch,'facevertexcdata',cd);
            counter = counter + 1;
    end
end

% Locate the figure(s) with the best CH and Silhouette measures
% CH index -> 'purple box'
hf = subplot(nrow, ncol, CHindex);
hf = get(hf, 'Position');
annotation('rectangle', hf, 'EdgeColor', [0.5,0,0.5], 'LineWidth', 5.0);

% Locate the figure(s) with the best G measure
% G index -> 'purple box'
hf = subplot(nrow, ncol, Gindex);
hf = get(hf, 'Position');
annotation('rectangle', hf, 'EdgeColor', 'Red', 'LineWidth', 5.0);

% Silhouette index -> 'green box'
hs = subplot(nrow, ncol, Sindex);
hs = get(hs, 'Position');
annotation('rectangle', hs, 'EdgeColor', 'g', 'LineWidth', 5.0);
end
```

```
==================
2. createResults.m
==================
    % createResukts: Creates the CH indices, Silhouette Averages, Gap
    %                statistics and the NCut segmentation display objects
    %
    %   createEesults outputs a matrix of CH indices, Silhoette Average,
    %   and Gap statistic for each NCut segmentation with different Number
    %   of Segments and Neighborhood Radius. Furthermore, this function
    %   returns two matrices of NCut segmentation display objects, with
    %   lines and colored, for each Number of Segment and Neighborhood Radius.

function [testStatMat_CH, testStatMat_S, testStatMat_G, segObj_array, colObj_array] ...
    = createResults(numSegs,sigmamatRad,dataW,sigmamatParam,imgFile)

% nrow - Number of Semgents
% ncol - Neighborhood Radiuss
nrow = numSegs(2)-numSegs(1)+1;
ncol = sigmamatRad(2)-sigmamatRad(1)+1;

% Matrices for each evaluation measure
testStatMat_CH = zeros(nrow,ncol);
testStatMat_S = cell(nrow,ncol);
testStatMat_G = zeros(nrow,ncol);

% Matrices of NCUt display objects
segObj_array = cell(nrow,ncol);
colObj_array = cell(nrow,ncol);

for i = 1:nrow
    nbSegments = numSegs(1)+i-1;

     %% Shape Tuning Method (supervised)
     cd('C:\Documents and Settings\Yi Xiang Chong\My Documents\MATLAB Workspace\
NcutImage_Original\NcutImage_7\');
     main;
     SegLabel_prior = demoNcutImage(nbSegments, imgFile);
     cd('C:\Documents and Settings\Yi Xiang Chong\My Documents\MATLAB Workspace\
NcutImage_7_1\NcutImage_7\');
     main;
```

```
    for j = 1:ncol
        sigmamatParam.RADIUS = sigmamatRad(1)+j-1;

        % Run the NCut Algorithm
        [segObj, colObj, CH, S, G] = runNcutImage(nbSegments, dataW, sigmamatParam,
 SegLabel_prior, imgFile);

        % Building the Test Stat Matrix
        testStatMat_CH(i,j) = CH;
        testStatMat_G(i,j) = G;
        testStatMat_S{i,j} = S;

        % Collecting the display objects
        segObj_array{i,j} = segObj;
        colObj_array{i,j} = colObj;
    end
end

end
```

```
====================
3. runNcutImage.m
====================
```

```
%%
    % runNCutImage: Runs the NCut algorithm to create segmentation display
    %               objects and the evaluation measures
    %
    % Run demoNcut Image without displaying figure or user interaction
    % Run demoNuct by varying these following parameters:
        % nbSegments - number of Segments
        % sampleRadius - sampling Radius
        % sampleRate - sampling Rate
        % edgeVariance - constant locality variance
        % SegLabel_prior - regions from original NCut segmentation
        % imgFile = image file name

    % Output image objects and evaluation measures
%%

function [segObj, colObj, CH, S, G] = runNcutImage(nbSegments, dataW, sigmamatParam,
 SegLabel_prior, imgFile);
```

```
main;

%% read image, change color image to brightness image, resize to 160x160
I = imread_ncut(['specific_NcutImage_files/jpg_images/' imgFile],160,160);

% Add random noise if necessary for robustness test
% I = I + normrnd(0,20,size(I));

%% compute the edges imageEdges, the similarity matrix W based on
%% Intervening Contours, the Ncut eigenvectors and discrete segmentation
disp('computing Ncut eigenvectors ...');
tic;
[SegLabel,NcutDiscrete,NcutEigenvectors,NcutEigenvalues,W,imageEdges] = ...
    NcutImage(I,nbSegments,dataW,sigmamatParam,SegLabel_prior);
disp(['The computation took ' num2str(toc) ' seconds on the ' num2str(size(I,1)) 'x'
num2str(size(I,2)) ' image']);

%% save the segmentation display objects
bw = edge(SegLabel,0.01);

J1=showmask(I,imdilate(bw,ones(2,2)));
J2=showmask(SegLabel,imdilate(bw,ones(2,2)));

segObj = J1;
colObj = J2(:,:,1);
disp('This is the segmentation');

%% Compute the evaluation measures

% Declaring the necessary variables and arrays
CLUSTER = cell(1,nbSegments);
CLUSTER_unif = cell(1,nbSegments);
WCSS = zeros(1,nbSegments);
WCSS_unif = zeros(1,nbSegments);
BCSS = zeros(1,nbSegments);
SCALE = max(I(:));

I_new = I ./ SCALE;
I_unif = I(randperm(size(I,1)), randperm(size(I,2)));

elements = unique(SegLabel);
for i=1:nbSegments
```

```
   CLUSTER{i} = I_new(SegLabel==elements(i));
   CLUSTER_unif{i} = I_unif(SegLabel==elements(i));
end

% Within cluster sum of squares (WCSS) for each cluster
for i = 1:nbSegments
    temp = CLUSTER{i};
    temp2 = CLUSTER_unif{i};
    % var normalizes V by N  1
    WCSS(i) = var(temp(:))*(length(temp(:))-1);
    WCSS_unif(i) = var(temp2(:))*(length(temp2(:))-1);
end

% Between clusters sum of squares (BCSS) for each cluster
for i = 1:nbSegments
     grand_avg = mean(I_new(:));
     temp = CLUSTER{i};
     clust_avg = mean(temp(:));

     BCSS(i) = length(CLUSTER{i}) * (clust_avg - grand_avg)^2;
end

%%  Compute the CH index, CH
WCSS_tot = sum(WCSS);
WCSS_tot_unif = sum(WCSS_unif);
BCSS_tot = sum(BCSS);

CH = (BCSS_tot/(nbSegments-1)) / (WCSS_tot/(length(I(:))-nbSegments));

%% Compute the Gap Statistic, G

G = log(WCSS_tot) - log(WCSS_tot_unif);

%% Compute the test statistic S, "Silhouette Coefficient"

% Declaring the necessary variables and arrays
TESTSTAT_S = cell(1,nbSegments);

Svalues = silhouette(I_new(:),SegLabel(:),'sqeuclid');

for i=1:nbSegments
    TESTSTAT_S{1,i} = Svalues(SegLabel(:)==elements(i));
```

```
end

S = TESTSTAT_S;

disp('The demo is finished.');
end

==================
4. NcutImage.m
==================
%  [SegLabel,NcutDiscrete,NcutEigenvectors,NcutEigenvalues,W,imageEdges]=
NcutImage(I);
%  Input: I = brightness image
%         nbSegments = number of segmentation desired
%  Output: SegLable = label map of the segmented image
%          NcutDiscrete = Discretized Ncut vectors
%
% Timothee Cour, Stella Yu, Jianbo Shi, 2004.

function [SegLabel,NcutDiscrete,NcutEigenvectors,NcutEigenvalues,W,imageEdges]=
 NcutImage(I,nbSegments,dataW,sigmamatParam,SegLabel_prior)

if nargin <2,
   nbSegments = 10;
end

% Return W = matrix of edges weight (similarity matrix)
%        imageEdges = image showing edges extracted

[W,imageEdges] = ICgraph(I,sigmamatParam,nbSegments,SegLabel_prior,dataW);

[NcutDiscrete,NcutEigenvectors,NcutEigenvalues] = ncutW(W,nbSegments);

%% generate segmentation label map
[nr,nc,nb] = size(I);

SegLabel = zeros(nr,nc);
for j=1:size(NcutDiscrete,2),
    SegLabel = SegLabel + j*reshape(NcutDiscrete(:,j),nr,nc);
end
```

```
==================
5. ICGraph.m
==================
% [W,imageEdges] = ICgraph(I,dataW,dataEdgemap);
% Input:
% I = gray-level image
% optional parameters:
% dataW.sampleRadius=10;
% dataW.sample_rate=0.3;
% dataW.edgeVariance = 0.1;
%
% dataEdgemap.parametres=[4,3, 21,3];%[number of filter orientations, number of scales, filter size,
% dataEdgemap.threshold=0.02;
%
% Output:
% W: npixels x npixels similarity matrix based on Intervening Contours
% imageEdges: image showing edges extracted in the image
%
% Timothee Cour, Stella Yu, Jianbo Shi, 2004.

function [W,imageEdges] =
ICgraph(I,sigmamatParam,nbSegments,SegLabel_prior,dataW,dataEdgemap);

[p,q] = size(I);

if (nargin< 5) | isempty(dataW),
    dataW.sampleRadius=10;      % default was 10
    dataW.sample_rate=0.3;      % default was 0.3
    dataW.edgeVariance = 0.1;  % default was 0.1

    % 0.05 good segmentation for face images.
end

if (nargin<6) | isempty(dataEdgemap),
    dataEdgemap.parametres=[4,3, 21,3];%[number of filter orientations, number of scales,
filter size, elongation]
    dataEdgemap.threshold=0.02;
end

edgemap =
computeEdges(I,dataEdgemap.parametres,dataEdgemap.threshold,sigmamatParam,nbSegments,SegLabel_prior)
```

```
imageEdges = edgemap.imageEdges;
W =
computeW(I,dataW,edgemap.emag,edgemap.sigmamat,edgemap.ephase,sigmamatParam);

=======================
6. computeEdges.m
=======================
% edgemap = computeEdges(imageX,parametres,threshold)
%
% computes the edge in imageX with parameters parametres and threshold
% Timothee Cour, Stella Yu, Jianbo Shi, 2004.

function edgemap =
computeEdges(imageX,parametres,threshold,sigmamatParam,nbSegments,SegLabel_prior)

[ex,ey,egx,egy,eg_par,eg_th,emag,ephase , g ] =
quadedgep(imageX,parametres,threshold);
% example : [ex,ey,egx,egy,eg_par,eg_th,emag,ephase] = quadedgep(imageX,
[4,3,30,3],0.05);

% function [x,y,gx,gy,par,threshold,mag,mage,g,FIe,FIo,mago] =
quadedgep(I,par,threshold);
% Input:
%    I = image
%    par = vector for 4 parameters
%      [number of filter orientations, number of scales, filter size, elongation]
%      To use default values, put 0.
%    threshold = threshold on edge strength
% Output:
%    [x,y,gx,gy] = locations and gradients of an ordered list of edgels
%       x,y could be horizontal or vertical or 45 between pixel sites
%       but it is guaranteed that there [floor(y) + (floor(x)-1)*nr]
%       is ordered and unique.  In other words, each edgel has a unique pixel id.
%    par = actual par used
%    threshold = actual threshold used
%    mag = edge magnitude
%    mage = phase map
%    g = gradient map at each pixel
%    [FIe,FIo] = odd and even filter outputs
%    mago = odd filter output of optimum orientation
%
% Stella X. Yu, 2001
```

```
% [emagTrie,eindex] = sort(emag);

%edges3 = sparse(floor(ex),floor(ey),
(egx.^2+egy.^2).^(1/2),size(imageX,2),size(imageX,1))';

%% To compute sigma_ij matrix (Added Yi Xiang 11/10/2011)

if (sigmamatParam.YES==1)

    [row,col] = size(emag);
    varmat = zeros(row,col);
    rad = sigmamatParam.RADIUS;

    for i=1:row
        for j=1:col
            % temp = ngbhdpoints(i,j,emag,rad);
            % varmat(i,j) = var(temp(:));   % using pearson variance
            % varmat(i,j) = distanceCov(temp(:),temp(:)); % using distance variance
            varmat(i,j) = decaying_ngbhd_var(i,j,emag,rad); % using decaying variance
        end
    end

    sigmamat = sqrt(varmat);

    % HeatMap(flipud(sigmamat));

    %% Canny Enhanced Method
    % temp = edge(imageX,'canny');
    % sigmamat_CANNY = sigmamat;


    %% Shape Tuning Method
    sigmamat_CLUSTERS = cell(1,nbSegments);
    elements = unique(SegLabel_prior);
    for i=1:nbSegments
        sigmamat_CLUSTERS{i} = sigmamat(SegLabel_prior==elements(i));
    end

    % Taking the mode
    sigmamat_new = sigmamat;
```

```
    for i = 1:nbSegments
       sigmamat_new(SegLabel_prior==elements(i)) = mode(sigmamat_CLUSTERS{i});
    end

     %% Box Bounding Method
%     A = SegLabel_prior==elements(i);
%         [rB, rT, cL, cR, B] = truncateMatrix(A);
%         if(rB~=0)
%             if(cL==0) A(rB,1:cR) = A(rB,1:cR) | 1;
%             elseif (cR==0) A(rB,cL:size(A,2)) = A(rB,cL:size(A,2)) | 1;
%             else A(rB,cL:cR) = A(rB,cL:cR) | 1;
%             end
%         end
%
%         if(rT~=0)
%             if(cL==0) A(rT,1:cR) = A(rT,1:cR) | 1;
%             elseif (cR==0) A(rT,cL:size(A,2)) = A(rT,cL:size(A,2)) | 1;
%             else A(rT,cL:cR) = A(rT,cL:cR) | 1;
%             end
%         end
%
%         if(cL~=0)
%             if(rT==0) A(1:rB,cL) = A(1:rB,cL) | 1;
%             elseif (rB==0) A(rT:size(A,1),cL) = A(rT:size(A,1),cL) | 1;
%             else A(rT:rB,cL) = A(rT:rB,cL) | 1;
%             end
%         end
%
%         if(cR~=0)
%             if(rT==0) A(1:rB,cR) = A(1:rB,cR) | 1;
%             elseif (rB==0) A(rT:size(A,1),cR) = A(rT:size(A,1),cR) | 1;
%             else A(rT:rB,cR) = A(rT:rB,cR) | 1;
%             end
%         end
%           sigmamat_CLUSTERS{i} = sigmamat(A);

      %% Exponential Decay Method
%     for i = 1:nbSegments
%        %% New method
%        A = sigmamat .* (SegLabel_prior==elements(i));
%        A = truncateMatrix(A);
%        [rowA, colA] = size(A);
```

```
%
%        if (mod(A,2)==0) % if even
%            A = [A(1:rowA/2, 1:colA/2) NaN(rowA/2,1) A(1:rowA/2, colA/2+1:colA);
%                    NaN(1,colA+1);
%                    A(rowA/2+1:rowA, 1:colA/2) NaN(rowA/2,1) A(rowA/2+1:rowA,
colA/2+1:colA)];
%            center_r = rowA/2+1;
%            center_c = colA/2+1;
%
%            sigmamat_new(SegLabel_prior==elements(i)) = ...
%                (decaying_ngbhd_var(center_r,center_c,A,rowA/2));
%
%        else % if odd
%            center_r = floor(rowA/2)+1;
%            center_c = floor(colA/2)+1;
%
%            sigmamat_new(SegLabel_prior==elements(i)) = ...
%                (decaying_ngbhd_var(center_r,center_c,A,rowA/2));
%
%        end
%   end

    % HeatMap(sigmamat_new)
    % sigmamat_new(find(temp)) = sigmamat(find(temp));
    sigmamat = sigmamat_new;
    % sigmamat = sigmamat_CANNY;

end

%%
try
    temp = edge(imageX,'canny');
    edges2 = emag .* temp ;
    if (sigmamatParam.CANNY == 1)
        sigmamat = sigmamat .* temp;
        % HeatMap(flipud(sigmamat));
    end
    %edges2 = emag .* edge(imageX,'sobel') ;
catch
    edges2 = 0 * emag;
end
```

```
edges2 = edges2 .* (edges2 > threshold);
egx1 = g(:,:,1);
egy1 = g(:,:,2);
eindex = find(edges2);
[ey,ex,values] = find(edges2);

egx = egx1(eindex);
egy = egy1(eindex);

%% Include sigmamat in edgemap (Added by Yi Xiang 11/10/2011)
if (sigmamatParam.YES==1)
    edgemap.sigmamat = sigmamat;
else
    edgemap.sigmamat = NaN;
end

edgemap.eindex = eindex;
edgemap.values = values;
edgemap.x = ex;
edgemap.y = ey;
edgemap.gx = egx;
edgemap.gy = egy;
edgemap.emag = emag;
edgemap.ephase = ephase;
edgemap.imageEdges = edges2;

==================
7. computeW.m
==================
% W = computeW(imageX,dataW,emag,ephase)
% Timothee Cour, Stella Yu, Jianbo Shi, 2004.

% * function [i,j] = cimgnbmap([nr,nc], nb_r, sample_rate)
% *   computes the neighbourhood index matrix of an image,
% *   with each neighbourhood sampled.
% * Input:
% *   [nr,nc] = image size
% *   nb_r = neighbourhood radius, could be [r_i,r_j] for i,j
% *   sample_rate = sampling rate, default = 1
% * Output:
% *   [i,j] = each is a column vector, give indices of neighbour pairs
% *     UINT32 type
```

```
% *       i is of total length of valid elements, 0 for first row
% *       j is of length nr * nc + 1

function W = computeW(imageX,dataW,emag,sigmamat,ephase,sigmamatParam)

[p,q] = size(imageX);

[w_i,w_j] = cimgnbmap([p,q],dataW.sampleRadius,dataW.sample_rate);

%% Added by Yi Xiang Chong
if (sigmamatParam.YES == 1)


    %% Look at the affinity matrix structure of W before normalizing by sigmamat
    % W = affinityic(emag,ephase,w_i,w_j,max(emag(:)) * dataW.edgeVariance);

    % spy(W)   % Note: Takes up a lot of processing time


    %% There are 3 methods (for now) of normalizing the affinity matrix
    %% Method 1: Normalizing intensity matrix first before
    %%           computing affinity matrix

    % emag = emag .* (sigmamat);
    % W_after_1 = affinityic(emag,ephase,w_i,w_j,max(emag(:)) * dataW.edgeVariance);

    %% Method 2: Normalizing affinity matrix with "sigma(i) * sigma(j)"
    %%           (default method in paper)

    W_after_2 = affinityic_2(emag,ephase,w_i,w_j,sigmamat);

    %% Method 3: Normalizing affinity matrix with "sigma(i) + sigma(j)"

    W_after_3 = affinityic_3(emag,ephase,w_i,w_j,sigmamat);

%% Pick the appropriate method
% W = W_after_1;  % Method 1
W = W_after_2;  % Method 2
% W = W_after_3;  % Method 3

else
    W = affinityic_ori(emag,ephase,w_i,w_j,max(emag(:)) * dataW.edgeVariance);
```

```
end

W = W/max(W(:));

=================
8. affinityic.c
=================

/*=================================================================
* function w = affinityic(emag,ephase,pi,pj,sigma)
* Input:
*   emag = edge strength at each pixel
*   ephase = edge phase at each pixel
*   [pi,pj] = index pair representation for MALTAB sparse matrices
*   sigma = sigma for IC energy
* Output:
*   w = affinity with IC at [pi,pj]
*

% test sequence
f = synimg(10);
[i,j] = cimgnbmap(size(f),2);
[ex,ey,egx,egy] = quadedgep(f);
a = affinityic(ex,ey,egx,egy,i,j)
show_dist_w(f,a);

* Stella X. Yu, Nov 19, 2001.
*=================================================================*/

# include "mex.h"
# include "math.h"

void mexFunction(
    int nargout,
    mxArray *out[],
    int nargin,
    const mxArray *in[]
)
{
    /* declare variables */
    int nr, nc, np, total;
```

```
    int i, j, k, ix, iy, jx, jy, ii, jj, iip1, jjp1, iip2, jjp2, step;
    double sigma, di, dj, a, z, maxori, phase1, phase2, slope;
int *ir, *jc;
unsigned long *pi, *pj;
double *emag, *ephase, *w;

    /* check argument */
    if (nargin<4) {
        mexErrMsgTxt("Four input arguments required");
    }
    if (nargout>1) {
        mexErrMsgTxt("Too many output arguments");
    }

    /* get edgel information */
nr = mxGetM(in[0]);
nc = mxGetN(in[0]);
if ( nr*nc ==0 || nr != mxGetM(in[1]) || nc != mxGetN(in[1]) ) {
    mexErrMsgTxt("Edge magnitude and phase shall be of the same image size");
}
    emag = mxGetPr(in[0]);
    ephase = mxGetPr(in[1]);
    np = nr * nc;

    /* get new index pair */
    if (!mxIsUint32(in[2]) | !mxIsUint32(in[3])) {
        mexErrMsgTxt("Index pair shall be of type UINT32");
    }
    if (mxGetM(in[3]) * mxGetN(in[3]) != np + 1) {
        mexErrMsgTxt("Wrong index representation");
    }
    pi = mxGetData(in[2]);
    pj = mxGetData(in[3]);

    /* create output */
    out[0] = mxCreateSparse(np,np,pj[np],mxREAL);
    if (out[0]==NULL) {
    mexErrMsgTxt("Not enough memory for the output matrix");
}
w = mxGetPr(out[0]);
ir = mxGetIr(out[0]);
jc = mxGetJc(out[0]);
```

```
    /* find my sigma */
if (nargin<5) {
    sigma = 0;
     for (k=0; k<np; k++) {
         if (emag[k]>sigma) { sigma = emag[k]; }
     }
     sigma = sigma / 6;
     printf("sigma = %6.5f",sigma);
} else {
    sigma = mxGetScalar(in[4]);
}
//a = 0.5 / (sigma * sigma);

    /* computation */
    total = 0;
    for (j=0; j<np; j++) {

        jc[j] = total;
        jx = j / nr; /* col */
        jy = j % nr; /* row */

        for (k=pj[j]; k<pj[j+1]; k++) {

            i = pi[k];

            if (i==j) {
                maxori = 1;

            } else {

                ix = i / nr;
                iy = i % nr;

                /* scan */
                di = (double) (iy - jy);
                dj = (double) (ix - jx);

                maxori = 0.;
            phase1 = ephase[j];
```

```
    /* sample in i direction */
    if (abs(di) >= abs(dj)) {
     slope = dj / di;
     step = (iy>=jy) ? 1 : -1;

       iip1 = jy;
     jjp1 = jx;


    for (ii=0;ii<abs(di);ii++){
        iip2 = iip1 + step;
        jjp2 = (int)(0.5 + slope*(iip2-jy) + jx);

        phase2 = ephase[iip2+jjp2*nr];

        if (phase1 != phase2) {
            z = (emag[iip1+jjp1*nr] + emag[iip2+jjp2*nr]);
            if (z > maxori){
                maxori = z;
            }
        }

        iip1 = iip2;
        jjp1 = jjp2;
        phase1 = phase2;
    }

/* sample in j direction */
    } else {
    slope = di / dj;
    step =  (ix>=jx) ? 1: -1;

     jjp1 = jx;
    iip1 = jy;


    for (jj=0;jj<abs(dj);jj++){
        jjp2 = jjp1 + step;
        iip2 = (int)(0.5+ slope*(jjp2-jx) + jy);

        phase2 = ephase[iip2+jjp2*nr];
```

```
                    if (phase1 != phase2){
                        z = (emag[iip1+jjp1*nr] + emag[iip2+jjp2*nr]);
                        if (z > maxori){
                            maxori = z;
                        }

                    }

                    iip1 = iip2;
                    jjp1 = jjp2;
                    phase1 = phase2;
                }
                }

                //maxori = 0.5 * maxori;
                maxori = maxori;
                maxori = exp(-maxori * maxori);
                //maxori = exp(-maxori * maxori * a);
            }
    ir[total] = i;

    w[total] = maxori;
    total = total + 1;

} /* i */
    } /* j */

    jc[np] = total;
}
```

```
=================
9. affinityic_2.c
=================
```

```
/*===================================================================
* Modified to accomodate for 'sigma(i) * sigma(j)* method
*
* function w = affinityic_2(emag,ephase,pi,pj,sigmamat)
* Input:
*   emag = edge strength at each pixel
*   ephase = edge phase at each pixel
*   [pi,pj] = index pair representation for MALTAB sparse matrices
```

```
*   sigma = sigma for IC energy
* Output:
*   w = affinity with IC at [pi,pj]
*

% test sequence
f = synimg(10);
[i,j] = cimgnbmap(size(f),2);
[ex,ey,egx,egy] = quadedgep(f);
a = affinityic(ex,ey,egx,egy,i,j)
show_dist_w(f,a);

* Stella X. Yu, Nov 19, 2001.
*=================================================================*/

# include "mex.h"
# include "math.h"

void mexFunction(
    int nargout,
    mxArray *out[],
    int nargin,
    const mxArray *in[]
)
{
    /* declare variables */
    int nr, nc, np, total;
    int i, j, k, ix, iy, jx, jy, ii, jj, iip1, jjp1, iip2, jjp2, step;
    double sigma, di, dj, a, z, s, maxori, sMaxori, phase1, phase2, slope;
int *ir, *jc;
unsigned long *pi, *pj;
double *emag, *ephase, *w, *sigmamat;

    /* check argument */
    if (nargin<4) {
        mexErrMsgTxt("Four input arguments required");
    }
    if (nargout>1) {
        mexErrMsgTxt("Too many output arguments");
    }

    /* get edgel information */
```

```
nr = mxGetM(in[0]);
nc = mxGetN(in[0]);
if ( nr*nc ==0 || nr != mxGetM(in[1]) || nc != mxGetN(in[1]) ) {
    mexErrMsgTxt("Edge magnitude and phase shall be of the same image size");
}
    emag = mxGetPr(in[0]);
    ephase = mxGetPr(in[1]);
    np = nr * nc;

    /* get new index pair */
    if (!mxIsUint32(in[2]) | !mxIsUint32(in[3])) {
        mexErrMsgTxt("Index pair shall be of type UINT32");
    }
    if (mxGetM(in[3]) * mxGetN(in[3]) != np + 1) {
        mexErrMsgTxt("Wrong index representation");
    }
    pi = mxGetData(in[2]);
    pj = mxGetData(in[3]);

    /* create output */
    out[0] = mxCreateSparse(np,np,pj[np],mxREAL);
    if (out[0]==NULL) {
    mexErrMsgTxt("Not enough memory for the output matrix");
}
w = mxGetPr(out[0]);
ir = mxGetIr(out[0]);
jc = mxGetJc(out[0]);

    /* find my sigma */
if (nargin<5) {
    sigma = 0;
     for (k=0; k<np; k++) {
         if (emag[k]>sigma) { sigma = emag[k]; }
     }
     sigma = sigma / 6;
     printf("sigma = %6.5f",sigma);
} else {
    sigmamat = mxGetPr(in[4]);
}
//a = 0.5 / (sigma * sigma);

    /* computation */
```

```
total = 0;
for (j=0; j<np; j++) {

    jc[j] = total;
    jx = j / nr; /* col */
    jy = j % nr; /* row */

    for (k=pj[j]; k<pj[j+1]; k++) {

        i = pi[k];

        if (i==j) {
            sMaxori = 1;

        } else {

            ix = i / nr;
            iy = i % nr;

            /* scan */
            di = (double) (iy - jy);
            dj = (double) (ix - jx);

            sMaxori = 0.;
            maxori = 0.;
        phase1 = ephase[j];


            /* sample in i direction */
            if (abs(di) >= abs(dj)) {
             slope = dj / di;
             step = (iy>=jy) ? 1 : -1;

               iip1 = jy;
             jjp1 = jx;


            for (ii=0;ii<abs(di);ii++){
                iip2 = iip1 + step;
                jjp2 = (int)(0.5 + slope*(iip2-jy) + jx);

                phase2 = ephase[iip2+jjp2*nr];
```

```
        if (phase1 != phase2) {
            z = (emag[iip1+jjp1*nr] + emag[iip2+jjp2*nr]);
                s = (sigmamat[iip1+jjp1*nr] * sigmamat[iip2+jjp2*nr]);
            if (z > maxori){
                maxori = z;
                    sMaxori = s;
            }
        }

        iip1 = iip2;
        jjp1 = jjp2;
        phase1 = phase2;
    }

/* sample in j direction */
    } else {
    slope = di / dj;
    step =  (ix>=jx) ? 1: -1;

     jjp1 = jx;
    iip1 = jy;


    for (jj=0;jj<abs(dj);jj++){
        jjp2 = jjp1 + step;
        iip2 = (int)(0.5+ slope*(jjp2-jx) + jy);

        phase2 = ephase[iip2+jjp2*nr];

        if (phase1 != phase2){
            z = (emag[iip1+jjp1*nr] + emag[iip2+jjp2*nr]);
                s = (sigmamat[iip1+jjp1*nr] * sigmamat[iip2+jjp2*nr]);
            if (z > maxori){
                maxori = z;
                    sMaxori = s;
            }

        }

        iip1 = iip2;
        jjp1 = jjp2;
```

```
                    phase1 = phase2;
                }
                }

                //maxori = 0.5 * maxori;
                maxori = maxori;
                sMaxori = exp(-maxori * maxori * sMaxori);
                //maxori = exp(-maxori * maxori * a);
            }
    ir[total] = i;

    w[total] = sMaxori;
    total = total + 1;

} /* i */
    } /* j */

    jc[np] = total;
}
```

```
====================
10. affinityic_3.c
====================
```

```
/*================================================================
* Modified to accomodate for 'sigma(i) + sigma(j)' method
*
* function w = affinityic_2(emag,ephase,pi,pj,sigmamat)
* Input:
*   emag = edge strength at each pixel
*   ephase = edge phase at each pixel
*   [pi,pj] = index pair representation for MALTAB sparse matrices
*   sigma = sigma for IC energy
* Output:
*   w = affinity with IC at [pi,pj]
*

% test sequence
f = synimg(10);
[i,j] = cimgnbmap(size(f),2);
[ex,ey,egx,egy] = quadedgep(f);
a = affinityic(ex,ey,egx,egy,i,j)
```

```
show_dist_w(f,a);

* Stella X. Yu, Nov 19, 2001.
*================================================================*/

# include "mex.h"
# include "math.h"

void mexFunction(
    int nargout,
    mxArray *out[],
    int nargin,
    const mxArray *in[]
)
{
    /* declare variables */
    int nr, nc, np, total;
    int i, j, k, ix, iy, jx, jy, ii, jj, iip1, jjp1, iip2, jjp2, step;
    double sigma, di, dj, a, z, s, maxori, sMaxori, phase1, phase2, slope;
int *ir, *jc;
unsigned long *pi, *pj;
double *emag, *ephase, *w, *sigmamat;

    /* check argument */
    if (nargin<4) {
        mexErrMsgTxt("Four input arguments required");
    }
    if (nargout>1) {
        mexErrMsgTxt("Too many output arguments");
    }

    /* get edgel information */
nr = mxGetM(in[0]);
nc = mxGetN(in[0]);
if ( nr*nc ==0 || nr != mxGetM(in[1]) || nc != mxGetN(in[1]) ) {
    mexErrMsgTxt("Edge magnitude and phase shall be of the same image size");
}
    emag = mxGetPr(in[0]);
    ephase = mxGetPr(in[1]);
    np = nr * nc;

    /* get new index pair */
```

```
    if (!mxIsUint32(in[2]) | !mxIsUint32(in[3])) {
        mexErrMsgTxt("Index pair shall be of type UINT32");
    }
    if (mxGetM(in[3]) * mxGetN(in[3]) != np + 1) {
        mexErrMsgTxt("Wrong index representation");
    }
    pi = mxGetData(in[2]);
    pj = mxGetData(in[3]);

    /* create output */
    out[0] = mxCreateSparse(np,np,pj[np],mxREAL);
    if (out[0]==NULL) {
    mexErrMsgTxt("Not enough memory for the output matrix");
}
w = mxGetPr(out[0]);
ir = mxGetIr(out[0]);
jc = mxGetJc(out[0]);

    /* find my sigma */
if (nargin<5) {
    sigma = 0;
     for (k=0; k<np; k++) {
         if (emag[k]>sigma) { sigma = emag[k]; }
     }
     sigma = sigma / 6;
     printf("sigma = %6.5f",sigma);
} else {
    sigmamat = mxGetPr(in[4]);
}
//a = 0.5 / (sigma * sigma);

    /* computation */
    total = 0;
    for (j=0; j<np; j++) {

        jc[j] = total;
        jx = j / nr; /* col */
        jy = j % nr; /* row */

        for (k=pj[j]; k<pj[j+1]; k++) {

            i = pi[k];
```

```
if (i==j) {
    sMaxori = 1;

} else {

    ix = i / nr;
    iy = i % nr;

    /* scan */
    di = (double) (iy - jy);
    dj = (double) (ix - jx);

    sMaxori = 0.;
    maxori = 0.;
phase1 = ephase[j];


    /* sample in i direction */
    if (abs(di) >= abs(dj)) {
     slope = dj / di;
     step = (iy>=jy) ? 1 : -1;

       iip1 = jy;
     jjp1 = jx;


    for (ii=0;ii<abs(di);ii++){
        iip2 = iip1 + step;
        jjp2 = (int)(0.5 + slope*(iip2-jy) + jx);

        phase2 = ephase[iip2+jjp2*nr];

        if (phase1 != phase2) {
            z = (emag[iip1+jjp1*nr] + emag[iip2+jjp2*nr]);
                s = (sigmamat[iip1+jjp1*nr] + sigmamat[iip2+jjp2*nr]);
            if (z > maxori){
                maxori = z;
                    sMaxori = s;
            }
        }
```

```
                iip1 = iip2;
                jjp1 = jjp2;
                phase1 = phase2;
            }

        /* sample in j direction */
            } else {
            slope = di / dj;
            step =  (ix>=jx) ? 1: -1;

             jjp1 = jx;
            iip1 = jy;


            for (jj=0;jj<abs(dj);jj++){
                jjp2 = jjp1 + step;
                iip2 = (int)(0.5+ slope*(jjp2-jx) + jy);

                phase2 = ephase[iip2+jjp2*nr];

                if (phase1 != phase2){
                    z = (emag[iip1+jjp1*nr] + emag[iip2+jjp2*nr]);
                        s = (sigmamat[iip1+jjp1*nr] + sigmamat[iip2+jjp2*nr]);
                    if (z > maxori){
                        maxori = z;
                            sMaxori = s;
                    }

                }

                iip1 = iip2;
                jjp1 = jjp2;
                phase1 = phase2;
            }
            }

            //maxori = 0.5 * maxori;
            maxori = maxori;
            sMaxori = exp(-maxori * maxori * sMaxori);
            //maxori = exp(-maxori * maxori * a);
        }
ir[total] = i;
```

```
    w[total] = sMaxori;
    total = total + 1;

} /* i */
    } /* j */

    jc[np] = total;
}
```

```
========================
11. decaying_ngbhd_var.m
========================
    % decaying_ngbhd_var: Compute the exponentially weighted variance
    %                      for a given neighborhoods
    %
    % This function is written to compute an "exponential weight" variance for
    % for a given neighborhood of points with a particular radius
    %
    % nbghd radius = radius (default is 1)
    %
    % Only works for square matrix nxn, where n > 3
    % (Added Yi Xiang 11/10/2011
    %
function [decay_var] = decaying_ngbhd_var(r,c,A,grandRad)
    % Normalizing constant (so weights add up to 1)
    denominator = sum(exp(-(1:grandRad)));
    n_groups = grandRad;
    n_total = sum(~isnan(A(:)));

    var_temp = 0;
    grand_Mean = nanmean(A(:));

    for i = 1:grandRad
        ring_i = outermost_ngbhd_ring(r,c,A,i);
        ring_i(isnan(ring_i)) = [];
        group_i = length(ring_i(:));
        weight_i = group_i/(n_total-1) * (exp(-i)/denominator);
        var_temp = var_temp + weight_i * sum((ring_i - grand_Mean).^2);
    end
    decay_var = var_temp;
end
```

```
=======================
12. ngbhdpoints.m
=======================
% ngbhdpoints: Extracts the neighborhood points from
%              a reference point given a specific radius
%
% This function is written to extract a submatrix of neighborhood points
% from a larger matrix A, given the reference point (r,c)
%
% nbghd radius = radius (default is 1)
%
% Only works for square matrix nxn, where n > 3
% (Added Yi Xiang 11/10/2011
%
function [ngbhdpoints] = ngbhdpoints(r,c,A,radius)
    [row,col] = size(A);

    % Up(U); Down(D); Left(L); Right(R) of circle radius
    % The 16 different combinations

    % Within bounds:
    %       U               D                   L                   R
    if ((r-radius)>=1 && (r+radius)<=row && (c-radius)>=1 && (c+radius)<=col)
        B = A(r-radius:r+radius, c-radius:c+radius); % neighborhood

    % Up(U) radius out of bounds
    elseif ((r-radius)<1 && (r+radius)<=row && (c-radius)>=1 && (c+radius)<=col)
        B = A(1:r+radius, c-radius:c+radius);
    % Down(D) radius out of bounds
    elseif ((r-radius)>=1 && (r+radius)>row && (c-radius)>=1 && (c+radius)<=col)
        B = A(r-radius:row, c-radius:c+radius);
    % Left(L) radius out of bounds
    elseif ((r-radius)>=1 && (r+radius)<=row && (c-radius)<1 && (c+radius)<=col)
        B = A(r-radius:r+radius, 1:c+radius);
    % Right(R) radius out of bounds
    elseif ((r-radius)>=1 && (r+radius)<=row && (c-radius)>=1 && (c+radius)>col)
        B = A(r-radius:r+radius, c-radius:col);

    % Up(U) and Down(D) radius out of bounds
    elseif ((r-radius)<1 && (r+radius)>row && (c-radius)>=1 && (c+radius)<=col)
        B = A(1:row, c-radius:c+radius);
```

```
    % Left(L) and Right(R) radius out of bounds
    elseif ((r-radius)>=1 && (r+radius)<=row && (c-radius)<1 && (c+radius)>col)
        B = A(r-radius:r+radius, 1:col);
    % Up(U) and Left(L) radius out of bounds
    elseif ((r-radius)<1 && (r+radius)<=row && (c-radius)<1 && (c+radius)<=col)
        B = A(1:r+radius, 1:c+radius);
    % Up(U) and Right(R) radius out of bounds
    elseif ((r-radius)<1 && (r+radius)<=row && (c-radius)>=1 && (c+radius)>col)
        B = A(1:r+radius, c-radius:col);
    % Down(D) and Left(L) radius out of bounds
    elseif ((r-radius)>=1 && (r+radius)>row && (c-radius)<1 && (c+radius)<=col)
        B = A(r-radius:row, 1:c+radius);
    % Down(D) and Right(R) radius out of bounds
    elseif ((r-radius)>=1 && (r+radius)>row && (c-radius)>=1 && (c+radius)>col)
        B = A(r-radius:row, c-radius:col);

    % Up(U) and Down(D) and Left(L) radius out of bounds
    elseif ((r-radius)<1 && (r+radius)>row && (c-radius)<1 && (c+radius)<=col)
        B = A(1:row, 1:c+radius);
    % Up(U) and Down(D) and Right(R) radius out of bounds
    elseif ((r-radius)<1 && (r+radius)>row && (c-radius)>=1 && (c+radius)>col)
        B = A(1:row, c-radius:col);
    % Left(L) and Right(R) and Down(D) radius out of bounds
    elseif ((r-radius)>=1 && (r+radius)>row && (c-radius)<1 && (c+radius)>col)
        B = A(r-radius:row, 1:col);
   % Left(L) and Right(R) and Up(U) radius out of bounds
    elseif ((r-radius)<1 && (r+radius)<=row && (c-radius)<1 && (c+radius)>col)
        B = A(1:r+radius, 1:col);

    % All: Up(U) and Down(D) and Left(L) and Right(R) out of bounds,
    %      just grab the whole matrix
    else
        B = A;

    end

    ngbhdpoints = B;

end
```

```
==============================
13. outermost_ngbhd_ring.m
==============================
% outermost_ngbhd_ring: Extract the outermost ring of points
%                       from a reference point given a radius
%
% This function returns a vector of the outermost ring of points
% from a larger matrix A, given the reference point (r,c)
%
% nbghd radius = radius (default is 1)
%
% Only works for square matrix nxn, where n > 3
% (Added Yi Xiang 11/10/2011
%
function [ring] = outermost_ngbhd_ring(r,c,A,radius)
    [row,col] = size(A);

    % Up(U); Down(D); Left(L); Right(R) of circle radius
    % The 16 different combinations

    % If radius == 1
    if (radius == 1)
        B = ngbhdpoints(r,c,A,radius);
    end
    % Within bounds:
    %       U               D                   L                   R
    if ((r-radius)>=1 && (r+radius)<=row && (c-radius)>=1 && (c+radius)<=col)
        B = [A(r-radius, c-radius:c+radius)  A(r+radius, c-radius:c+radius) ...
             A(r-radius:r+radius, c-radius)' A(r-radius:r+radius, c+radius)'];
        B = unique(B);

    % Up(U) radius out of bounds
    elseif ((r-radius)<1 && (r+radius)<=row && (c-radius)>=1 && (c+radius)<=col)
        B = [A(r+radius, c-radius:c+radius) ...
             A(1:r+radius, c-radius)' A(1:r+radius, c+radius)'];
        B = unique(B);
    % Down(D) radius out of bounds
    elseif ((r-radius)>=1 && (r+radius)>row && (c-radius)>=1 && (c+radius)<=col)
        B = [A(r-radius, c-radius:c+radius) ...
             A(r-radius:row, c-radius)' A(r-radius:row, c+radius)'];
        B = unique(B);
    % Left(L) radius out of bounds
```

```
 elseif ((r-radius)>=1 && (r+radius)<=row && (c-radius)<1 && (c+radius)<=col)
    B = [A(r-radius, 1:c+radius)  A(r+radius, 1:c+radius) ...
         A(r-radius:r+radius, c+radius)'];
    B = unique(B);
 % Right(R) radius out of bounds
 elseif ((r-radius)>=1 && (r+radius)<=row && (c-radius)>=1 && (c+radius)>col)
    B = [A(r-radius, c-radius:col)  A(r+radius, c-radius:col) ...
         A(r-radius:r+radius, c-radius)' ];
    B = unique(B);

 % Up(U) and Down(D) radius out of bounds
 elseif ((r-radius)<1 && (r+radius)>row && (c-radius)>=1 && (c+radius)<=col)
     B = [A(1:row, c-radius)' A(1:row, c+radius)'];
     B = unique(B);
 % Left(L) and Right(R) radius out of bounds
 elseif ((r-radius)>=1 && (r+radius)<=row && (c-radius)<1 && (c+radius)>col)
     B = [A(r-radius, 1:col)  A(r+radius, 1:col)];
     B = unique(B);
 % Up(U) and Left(L) radius out of bounds
 elseif ((r-radius)<1 && (r+radius)<=row && (c-radius)<1 && (c+radius)<=col)
    B = [A(r+radius, 1:c+radius) A(1:r+radius, c+radius)'];
    B = unique(B);
 % Up(U) and Right(R) radius out of bounds
 elseif ((r-radius)<1 && (r+radius)<=row && (c-radius)>=1 && (c+radius)>col)
    B = [A(r+radius, c-radius:col) A(1:r+radius, c-radius)'];
    B = unique(B);
 % Down(D) and Left(L) radius out of bounds
 elseif ((r-radius)>=1 && (r+radius)>row && (c-radius)<1 && (c+radius)<=col)
    B = [A(r-radius, 1:c+radius)  A(r-radius:row, c+radius)'];
    B = unique(B);
% Down(D) and Right(R) radius out of bounds
 elseif ((r-radius)>=1 && (r+radius)>row && (c-radius)>=1 && (c+radius)>col)
    B = [A(r-radius, c-radius:col) A(r-radius:row, c-radius)'];
    B = unique(B);

 % Up(U) and Down(D) and Left(L) radius out of bounds
 elseif ((r-radius)<1 && (r+radius)>row && (c-radius)<1 && (c+radius)<=col)
     B = A(1:row, c+radius)';
 % Up(U) and Down(D) and Right(R) radius out of bounds
 elseif ((r-radius)<1 && (r+radius)>row && (c-radius)>=1 && (c+radius)>col)
     B = A(1:row, c-radius)' ;
 % Left(L) and Right(R) and Down(D) radius out of bounds
```

```
    elseif ((r-radius)>=1 && (r+radius)>row && (c-radius)<1 && (c+radius)>col)
        B = A(r-radius, 1:col);
   % Left(L) and Right(R) and Up(U) radius out of bounds
    elseif ((r-radius)<1 && (r+radius)<=row && (c-radius)<1 && (c+radius)>col)
        B = A(r+radius, 1:col);

    % All: Up(U) and Down(D) and Left(L) and Right(R) out of bounds,
    %      just grab the edges of the whole matrix
    else
        B = [A(1, 1:col)  A(row, 1:col) A(1:row, 1)' A(1:row, col)'];
        B = unique(B);

    end

    ring = B;
end
```

```
=======================
14. truncateMatrix.m
=======================
% truncateMatrix: Returns the smallest square matrix with least number
%                 of zeros given a sparse matrix filled with zeros
%
% This function obtains the smallest submatrix with least number of
% zeros given a sparse matrix A
function [rBOTTOM, rTOP, cLEFT, cRIGHT, subMatrix] = truncateMatrix(A)

[row,col] = size(A);
borderCheck = 0; % check if it's border of submatrix

rBOTTOM = 0;
rTOP = 0;
cLEFT = 0;
cRIGHT = 0;

% For example
% 0 0 0 0
% 1 2 0 3
% 0 0 0 0
%
% So borderCheck = 1 (TRUE) when first row of
% zeros are removed.
```

```
% Remove rows of zeros from TOP
i = 1;
while (borderCheck == 0)
    if (sum(A(i,:))==0)
        A(i,:) = [];
        rTOP = i;
        i = i + 1;
        row = row - 1;
    else
        borderCheck = 1;
    end
end

% Remove rows of zeros from BOTTOM
borderCheck = 0;
i = row;
while (borderCheck == 0)
    if (sum(A(i,:))==0)
        A(i,:) = [];
        rBOTTOM = i;
        i = i - 1;
        row = row - 1;
    else
        borderCheck = 1;
    end
end

% Remove cols of zeros from LEFT
borderCheck = 0;
i = 1;
while (borderCheck == 0)
    if (sum(A(:,i))==0)
        A(:,i) = [];
        cLEFT = i;
        i = i + 1;
        col = col - 1;
    else
        borderCheck = 1;
    end
end
```

```
% Remove cols of zeros from RIGHT
borderCheck = 0;
i = col;
while (borderCheck == 0)
    if (sum(A(:,col))==0)
        A(:,i) = [];
        cRIGHT = i;
        i = i - 1;
        col = col - 1;
    else
        borderCheck = 1;
    end
end
subMatrix = A;
end
```