Acknowledgments

I would like to thank my mentors, Dhruv Batra and Adarsh Kowdle, for their guidance and support throughout the year. I would also like to thank Professor Tsuhan Chen and his research associate, Yao-Jen Chang, for giving me the opportunity to be a part of the Advanced Multimedia Processing (AMP) lab, and explore cutting-edge technology in the exciting field of Computer Vision.

Executive Summary

In this report, we look at the use of Local Color Pattern features in interactive co-segmentation. According to the Abstract, the goal was to additionally look at textual and DAISY features; however, due to time constraints, only Local Color Pattern features will be discussed.

The layout of the report is as follows. In Section 1, the problem of interactive co-segmentation, as well as its uses, will be described. Section 2 gives an overview of aspects of this project that were already in-place at the onset of my research; namely, a Graphical User Interface for Interactive Co-segmentation, as well as co-segmentation results for Luv color feature vectors. Section 3 will introduce the idea of a Local Color Pattern feature vector, and discuss its merits. Section 4 will present co-segmentation results of both Luv color and Local Color Pattern features. It will be seen that although the two features have similar co-segmentation accuracies under most conditions, there are some image sets for which Local Color Pattern performs much better. Finally, Section 5 will conclude the report by suggesting future directions for this project.

1 Introduction

As computers become increasingly powerful and ubiquitous in our lives, we would like to use them to automate mundane or tedious tasks. One of these tasks involves locating Objects-Of-Interest (OOIs) in a group of related images, and appears most notably in medical imaging, in the tagging of photos on Facebook, Flickr, and other image-hosting websites, and in the colorization of black-and-white films. In medical imaging, doctors look for anomalies (such as tumors) by looking for characteristic patterns or objects in a series of CT scans. The process of finding such objects is usually done subjectively by doctors or medical technicians, by looking at one image at a time. Similarly, photos in a Facebook album are tagged one photo at a time, and the process of coloring black-and-white films is done one frame at a time.

The process of locating OOIs in a set of images can therefore be extremely time-consuming and prone to error. This is particularly true in the film colorization example, where the graphics artist will be working with hundreds of thousands of frames, each frame containing many objects. With interactive co-segmentation, the same process can be done automatically, with just a few mouse clicks.

Co-segmentation can be described as the simultaneous segmentation of multiple images. Segmentation is the process of partitioning an image into multiple regions, sometimes called superpixels, such that the regions are similar with respect to some feature, such as color or texture. In interactive co-segmentation, the user inputs two sets of scribbles into an image in the image set. One set of scribbles defines the regions in the image that should be part of the foreground (the OOIs); the other set defines the regions that should be part of the background (everything else). These scribbles will then be used in conjunction with the extracted feature information to provide additional context as to the location of the OOIs in all the images.

As mentioned in the Abstract, an interactive GUI for performing co-segmentation already existed at the onset of this project. Thus, the focus of this report will not be on the co-segmentation process, but rather, on the evaluation of the performance of different features used in co-segmentation.

2 Previously-Established Results

There are two important programs/results that were already in existence prior to the onset of my research: a Graphical User Interface for Interactive Co-segmentation, as well as co-segmentation results for Luv color features, which were obtained from the GUI. Both will be discussed below.

A. Interactive Co-segmentation GUI

The GUI for Interactive Co-segmentation runs from Matlab, and code for performing the co-segmentation is also written in Matlab; however, the graphics portion of the program was coded in Java. Information on the various features of the GUI, as well as the steps for performing interactive co-segmentation, is listed in Appendix A.

B. Luv Color Feature

Before we can talk about specific features, we first need to define the concept of features in the context of computer vision. A feature can be defined as a piece of information that is relevant for solving a problem computationally. It does this by highlighting or emphasizing particular structures or patterns in the data. For instance, an edge is a feature used to describe sudden changes of intensity in an image. A binary matrix can be used to denote the existence or non-existence of an edge at a specific pixel.

Feature extraction is the process of representing data in such a way as to highlight certain features embedded in that data. Since features can be represented by vectors of different dimensions, feature extraction is often done by transformation of basis vectors. For example, most color-based representations of images are obtained from the RGB representation of that image. Once feature extraction has been done, the information obtained from features can be used in more complicated tasks, such as object recognition or co-segmentation.

The Luv color feature, which is 3-dimensional, is obtained simply by representing each pixel in the Luv color space. To distinguish this feature from the LCP features, which also use color, the Luv color feature will also be referred to as an "absolute color feature". Co-segmentation results for these features are presented later in the report.

3 The Local Color Pattern Feature

A. Description

As its name suggests, the Local Color Pattern (LCP) feature incorporates information about the color of a pixel in addition to the color of nearby pixels. More specifically, for each pixel *x* in the image, the LCP feature contains information about the following five pixels: a) *x*; and b) the four nearest pixels in the north, south, east, and west (NESW) cardinal directions that have a color that is different from that of *x*. In order to understand exactly what that statement means, a couple of questions must be addressed:

1) What does it mean to say that two pixels have different colors?
2) How are the relevant pixels represented in the LCP feature vector?

To answer the first question, we first need to define the terms *clustering* and *color mode*. *Clustering* is a process which attempts to partition a group of *n* points into *k* distinct groups, called *clusters* (usually *k*<<*n*). Since the data points being clustered are color space representations of pixels, the mean value of cluster *s* is known as the *color mode* of *s* with respect to the color space *C*. Furthermore, it can be said that points in cluster *s* belong to the color mode of *s*. Therefore, two pixels have different colors with respect to color space *C* if they belong to different color modes with respect to *C* [2].

To answer the second question, the relevant pixels are represented in the LCP feature vector according to their colors with respect to *C*. As an example, consider the following toy image shown below:

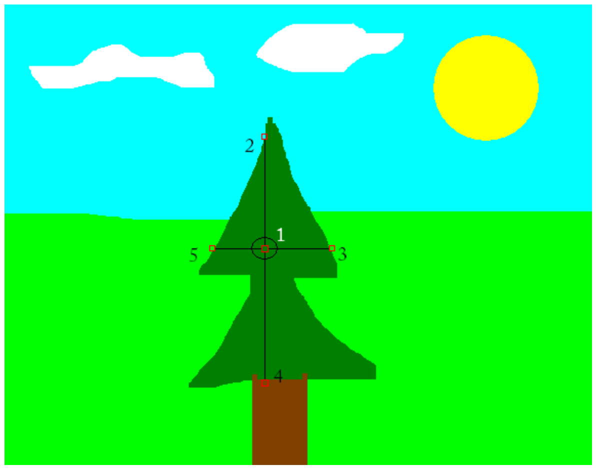


Figure 1: Computation of LCP Feature for One Pixel of an Image

The LCP feature vector for the highlighted pixel in Figure 1 would be given by the colors associated with points 1, 2, 3, 4, and 5, in that order. The order in which the pixels are represented is not important, as long as it is consistent for all pixels in all images in an image set. Since each point is represented by a 3-dimensional color space vector, the LCP feature vector is 15-dimensional.

B. Obtaining Local Color Pattern Features

The steps for obtaining LCP feature vectors for an image are as follows:

1) Convert image to desired color space.
2) Perform clustering to determine the color modes of the image. Assign a color mode label to each pixel in the image.
3) Perform a 4-way linear search over all pixels to find the nearest pixels in the NESW directions that belong to a different color mode.

For step 1, we will be looking at the RGB, HSV, and Luv color spaces. This step is done in Matlab; since the default representation for color images in Matlab is RGB, nothing needs to be done for this step if the desired color space is RGB. Appendix A has information on how to convert from the RGB color space to the other two color spaces, and vice versa.

For step 2, I tried a couple of software packages for clustering. At first, I used Charles Bouman's Gaussian Mixture Model (GMM) clustering package, which attempts to model the data as a linear combination of normal distributions with varying means and covariance matrices [1]. A good reference for GMM clustering, as well as the C code for performing GMM clustering, can be found here. However, I soon found that clustering using this package was far too slow, as it was taking around 20 minutes to cluster a single 300x500 image. I then tried Dan Pelleg's k-means clustering package [3], which was able to perform clustering of a similarly-sized image on the order of seconds. The k-means clustering package is run from Matlab using the `system` command.

For step 3, a MEX-file was created for doing the linear search in order to keep feature extraction runtime to a minimum. More efficient methods for performing this step could be considered in the future; however, with a multi-directional linear search, LCP features for a 300x500 pixel image could be obtained in under a second.

The code for performing steps 1 and 3 above is listed in Appendix B. The code for performing step 2 is provided in the references given above.

C. Motivation for Using Local Color Pattern Features

It can be seen that computing LCP vectors is far more resource-consuming than computing absolute color vectors, since calculation of LCP vectors requires clustering and vector searching in addition to color space conversion. So why use LCP?

It turns out that in many cases, looking solely at the colors of each pixel may not be enough to distinguish similarly-colored objects, if those objects don't both belong to background or foreground. For instance, consider an example image of a boy wearing a yellow shirt lying in the grass, with the sun overhead. If we consider the boy to be foreground, and everything else to be background, then using absolute color features alone (i.e. not taking into consideration neighboring pixels) would fail to put the boy and the sun in different categories. However, by using LCP features, we would be able to separate the two objects, since the boy is surrounded by grass, which is green, while the sun is surrounded by sky, which is blue.

4 Interactive Co-segmentation Results

A. Testing Methodology

Since the goal of co-segmentation is to automate a task that would normally be done by a human, the measure of success in co-segmentation for a particular image group would be to compare the co-segmentation results to a benchmark, which would be considered the "perfect" co-segmentation as perceived by a human observer. Qualitatively, this involves looking at each segmented image in an image group and seeing which pixels were marked correctly as foreground, and which pixels were not. Quantitatively, this involves computing the percentage of pixels that were correctly labeled as foreground or background.

Quantitative results can be obtained by first creating a black-and-white image for each image in the image set, where the white pixels correspond to pixels that should be marked foreground, and the black pixels correspond to pixels that should be marked background. This black-and-white image set is known as the *ground truth*, and is the baseline for comparison to the co-segmentation GUI results. Once the ground truth has been obtained, it is a simple matter to determine which pixels were marked correctly by co-segmentation, and which pixels were not.

One measure of accuracy of a particular feature can be obtained by counting the number of pixels marked correctly among all images in the image set, and dividing that number by the total number of pixels among all images in the image set. This gives an overall accuracy measure as a percent. We can obtain similar accuracies over the set of foreground and background pixels as well (as denoted in the ground truth). All of this information can be succinctly captured in a *confusion matrix*:

$$C = \begin{pmatrix} B_r & B_m \\ F_m & F_r \end{pmatrix} \tag{1}$$

where B_r represents the number of background pixels correctly marked as background, F_r represents the number of foreground pixels correctly marked as foreground, B_m represents the number of background pixels that should have been marked as foreground, and F_m represents the number of foreground pixels that should have been marked as background. Thus, we can obtain foreground, background, and overall accuracies as follows:

$$F_{acc} = \frac{F_r}{F_m + F_r} \tag{2}$$

$$B_{acc} = \frac{B_r}{B_m + B_r} \tag{3}$$

$$I_{acc} = \frac{F_r + B_r}{F_m + F_r + B_m + B_r} \tag{4}$$

It is important to consider background and foreground accuracies in addition to the overall accuracy, especially for image sets where the number of foreground or background pixels is a large percentage of the total number of pixels. For instance, consider an image set where only one-sixth of the total pixels represent foreground. If co-segmentation fails to find 50% of the foreground pixels, then the foreground accuracy is only 50%; however, the overall accuracy would be over 91%. Thus, by looking at overall accuracy alone, there might be situations where the overall accuracy is high, but the algorithm might have still failed to do a good job in finding the object-of-interest.

The co-segmentation GUI allows for scribbling on multiple images; however, only one image is marked with scribbles for each test run, to see how well the features perform with a minimal amount of information. Even so, there is some variation in this approach due to the fact that the scribbles will be different each time the GUI is run. However, this variation was reduced by scribbling on the same images for each image set, and by making the scribbles approximately the same for all trials.

B. Comparison of Absolute Color and Local Color Pattern Features

In this section, we will compare the co-segmentation performance of the Luv absolute color and LCP features for a group of one dozen different image sets. Out of these sets, 6 of them did relatively well with absolute color features, and the other 6 did relatively poorly.

Analysis of "Easy" Image Sets

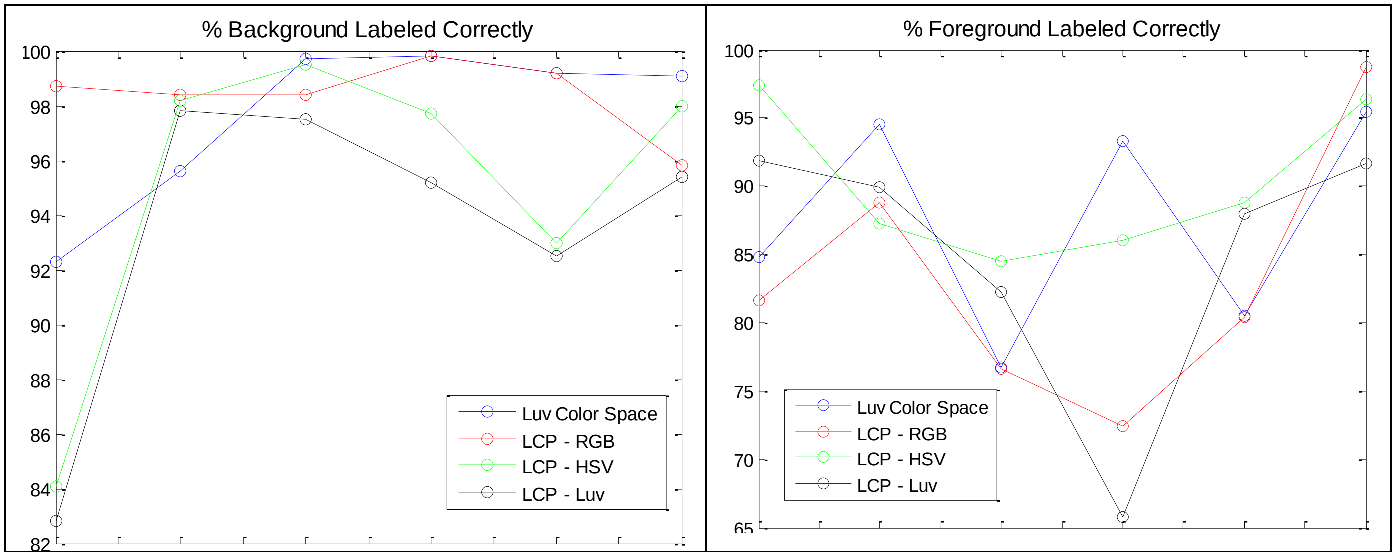
First, we will look at the performance of image sets where co-segmentation using Luv color features resulted in an accurate labeling of foreground and background pixels. In the following three figures, the percentage of background pixels, foreground pixels, and combined background and foreground pixels labeled correctly is plotted for each of the following six image sets,

017 – Taj Mahal 1	022 – Goose	025-1 – Airshows-Helicopter
032 – Brighton Kite Festival	040 – Monks	050 – Kendo (Helsinki)

using the following features:

- Luv color space
- LCP with respect to RGB, HSV, and Luv color spaces

On the horizontal axis, the points are plotted in order of increasing set number (i.e. starting from the Taj Mahal set and working up to the Kendo set).



Figures 2 and 3: Percentage of Background and Foreground Pixels Labeled Correctly – Easy Image Sets

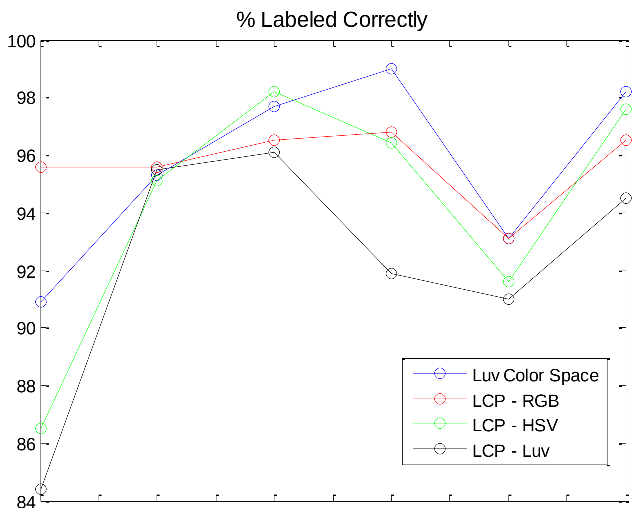


Figure 4: Overall Percentage of Pixels Labeled Correctly – Easy Image Sets

It can be seen that all four feature vectors perform relatively well, achieving over 90% accuracy on most or all of the background labels, over 80% accuracy on most of the foreground labels, and over 90% accuracy on overall labels. Local Color Pattern with respect to the RGB color space had the best overall and background labels accuracy, while Local Color Pattern with respect to the HSV color space did the best with respect to foreground labels.

The figures on the next page depict the segmentations for the Kendo image set for the Luv color and LCP with respect to HSV features. It can be seen that both segmentations are nearly perfect. A similarity between all of the high-accuracy sets is that the foreground colors and background colors are, for the most part, distinct, which means that a color feature can distinguish between the two even without contextual information from neighboring pixels.

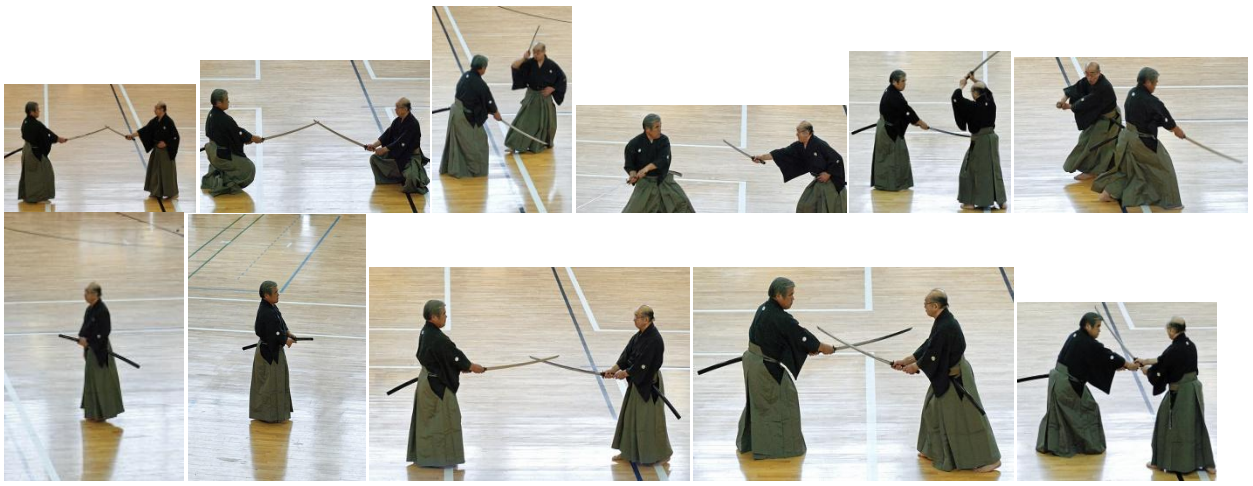


Figure 5: Kendo Image Set

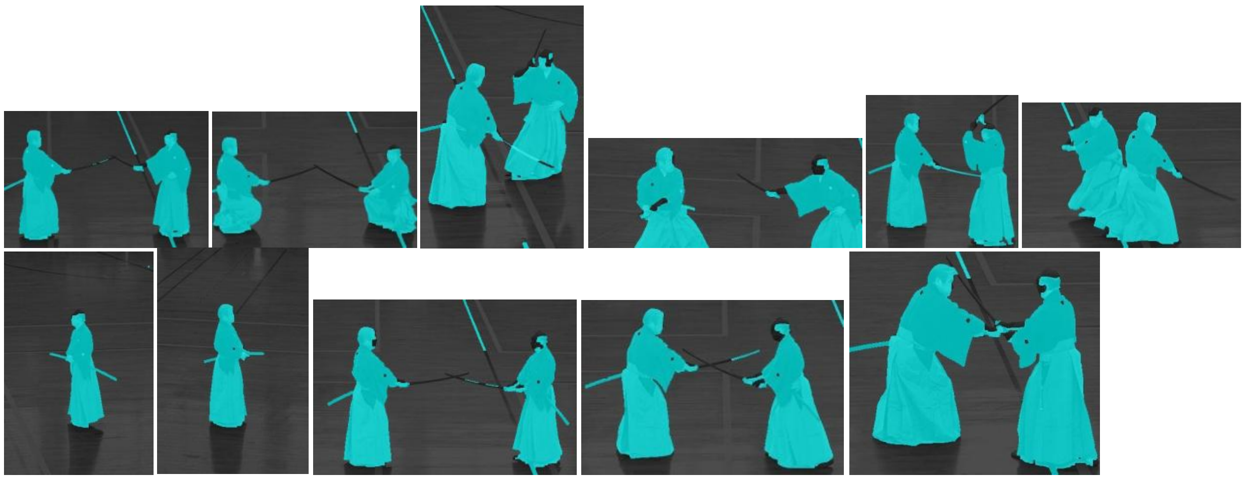


Figure 6: Kendo Image Set – Luv Color Feature Co-segmentation

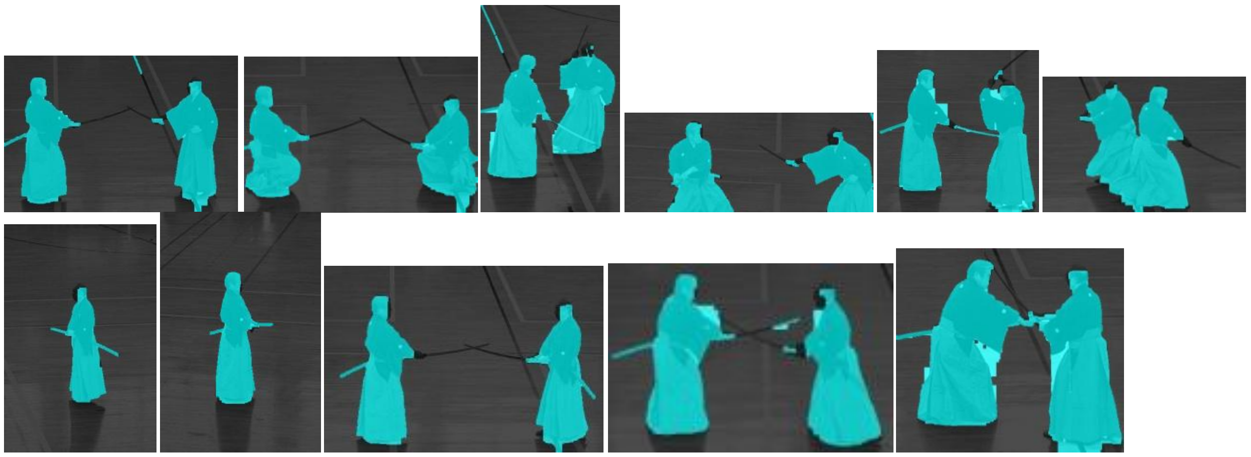
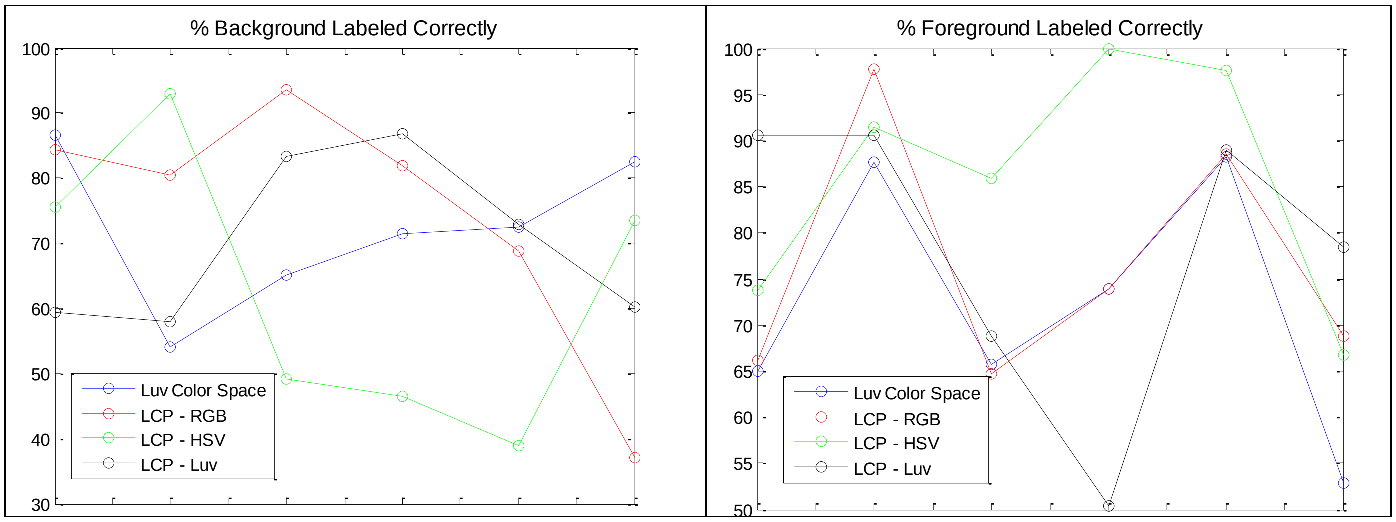


Figure 7: Kendo Image Set – LCP with Respect to HSV Co-segmentation

Analysis of “Hard” Image Sets

Next, we will look at the performance of image sets where co-segmentation using Luv color features resulted in an inaccurate labeling of foreground and background pixels. In the following three figures, the percentage of background pixels, foreground pixels, and combined background and foreground pixels labeled correctly is plotted for each of the following six image sets. As before, the same features are being tested, and points are plotted in order of increasing set number (i.e. starting from the Bear set and working up to the Panda set).

002 – Alaskan Brown Bear	009 – Stonehenge 1	012 – Stonehenge 2
020 – Pyramids	021 - Elephants	023 - Pandas



Figures 8 and 9: Percentage of Background and Foreground Pixels Labeled Correctly – Hard Image Sets

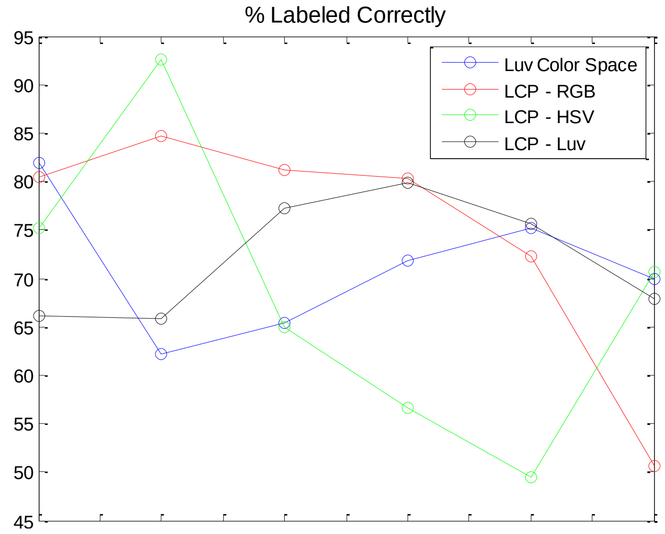


Figure 10: Overall Percentage of Pixels Labeled Correctly – Hard Image Sets

It can be seen that overall accuracies are much lower across all features. This is expected, since many of the images in these six sets have similar background and foreground colors. Consider an image from the Bear set, and an image from the Pyramid set, shown below.

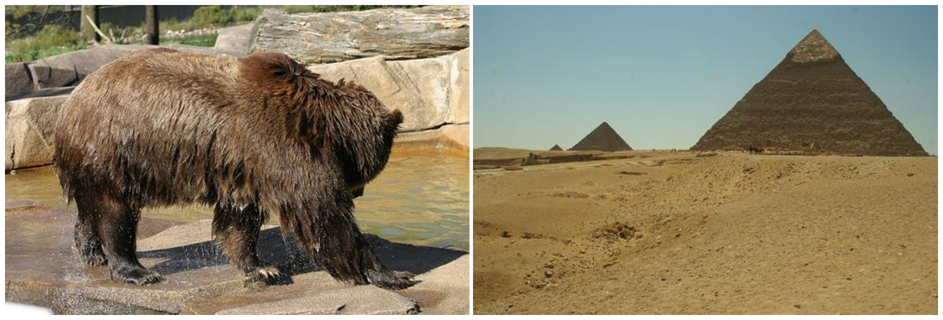


Figure 11: Left - Image from Bear set. Right – Image from Pyramid set.

If adjacent background and foreground regions are too similar in color, then points from both regions will fall into the same color mode, making accurate separation of foreground from background difficult. Interestingly enough, LCP with respect to HSV features had the best overall foreground labeling accuracies, just like it did with the “easy” image sets. Furthermore, for many of the sets where LCP-HSV did well in the foreground, it also did poorly in background labeling, which means that LCP-HSV tends to mark too many of the pixels as foreground.

However, one image set where LCP did particularly well when compared to the Luv color feature was in one of the Stonehenge sets. The figures on the next page depict the segmentations for the Stonehenge image set for the Luv color and LCP with respect to HSV features. It can be seen that the Luv color feature has trouble distinguishing between the grass and the stones. This is because the stones are not completely grey, but have tints of green in them, as can be seen below after zooming in on one of the images. Because some of the foreground scribbles cut through these green areas (in addition to touching the more grayish portions of stone), pretty much all the grassy areas were labeled as foreground when Luv color features were used, as can be seen in Figure 14.

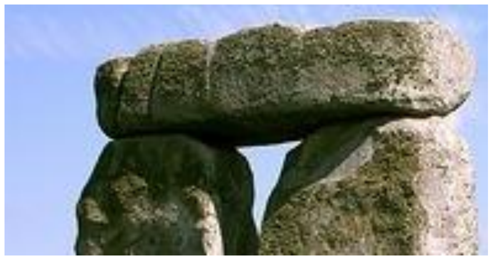


Figure 12: Stonehenge Image Set – Image 3, Zoomed-In

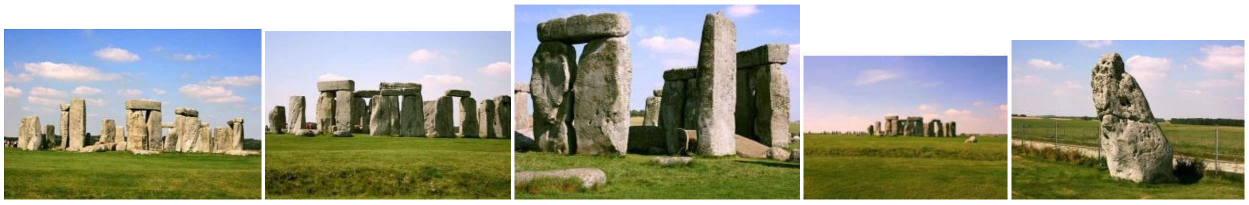


Figure 13: Stonehenge Image Set

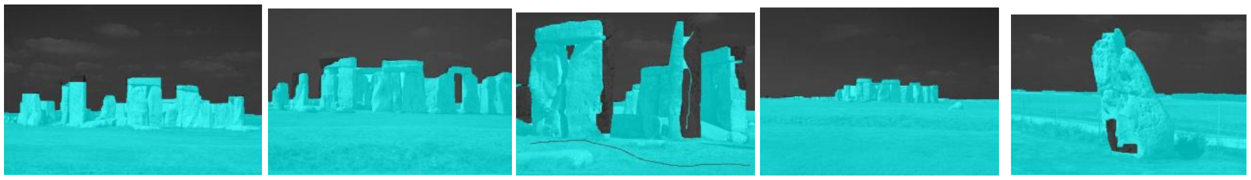


Figure 14: Stonehenge Image Set – Luv Color Feature Co-segmentation

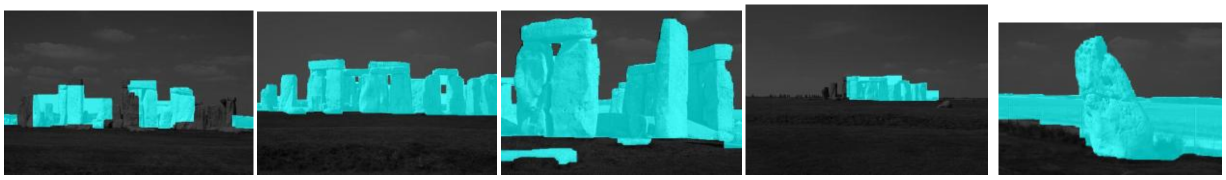


Figure 15: Stonehenge Image Set – LCP with Respect to HSV Co-segmentation

However, when the LCP-HSV feature was used, the majority of the grassy areas were correctly labeled as background, except for image 5 in the set, where part of the grass is highlighted. One possible explanation for this segmentation result is that the LCP feature picks up the greenish patches on the rocks, but only highlights green areas if they are touching the sky, which is bluish in color. In image 5, the patch of grass above the dirt road is touching the sky, so only that part of the grass is highlighted. In the other images, since there are stones between the grass and the sky, the grassy areas are correctly labeled as background.

5 Conclusion

In this report, the motivation behind the use of contextual information in feature representation for interactive co-segmentation was discussed. Specifically, the Local Color Pattern (LCP) feature, which uses contextual information with respect to various color spaces, was described in detail. A comparison of co-segmentation accuracy between LCP features and the already-implemented Luv absolute color features was done. It was found that while Local Color Pattern and absolute color features often produced similar results, LCP features could perform significantly better for certain image sets, such as the Stonehenge set that was discussed in the previous section.

If I were to continue work on this project, there would still be many things I could examine. For instance, I could look at using other types of features other than color, such as texture. The Bear image set, which did not have great co-segmentation accuracies using color features, should perform much better with textual features, since the bear (foreground) is the only object in the image that has a “furry” texture. I could also look into combining features. For instance, one of the features used in [1] combines LCP with edge information. There is still much that can be done with various types of features to improve co-segmentation accuracies to the point that we can reliably use co-segmentation to tag our photo albums on Facebook.

6 Works Cited

[1] Bouman, Charles A. “Cluster: An Unsupervised Algorithm for Modeling Gaussian Mixtures.”

[2] Cui, Jingyu et al. “Transductive Object Cutout.” *CVPR 2008: IEEE Conference on Computer Vision and Pattern Recognition* (2008).

[3] Pelleg, Dan. “K-means and X-means Implementations.” 13 Aug. 2004.

Appendix A: User's Guide for the Interactive Co-segmentation GUI

In order for co-segmentation to be performed, two things are needed: 1) an image set; and 2) feature representations of each image in the image set. Before the GUI can be loaded, the image folder must be specified in the GUI m-file (main_app.m). Additionally, a mat file containing the feature space representations of each image must be specified from within main_app.m. Once the image folder and mat file is specified, the GUI is run by typing in `main_app` on the command line. The following screen will appear:

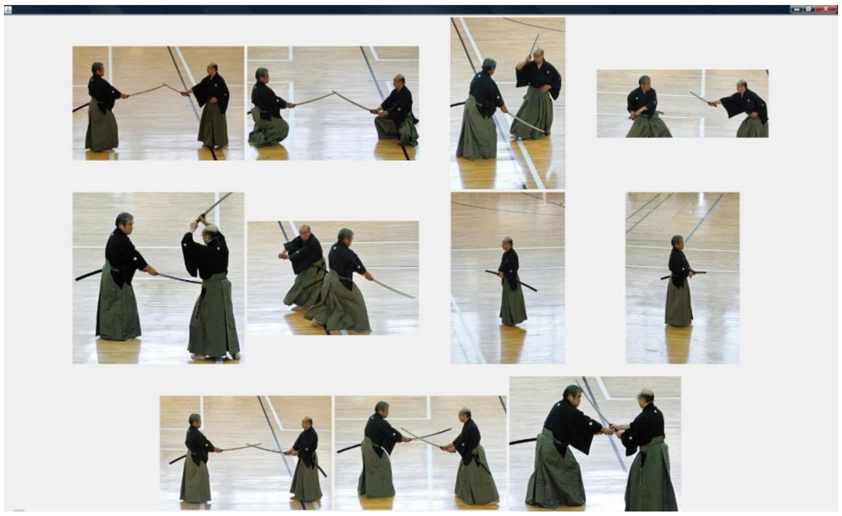
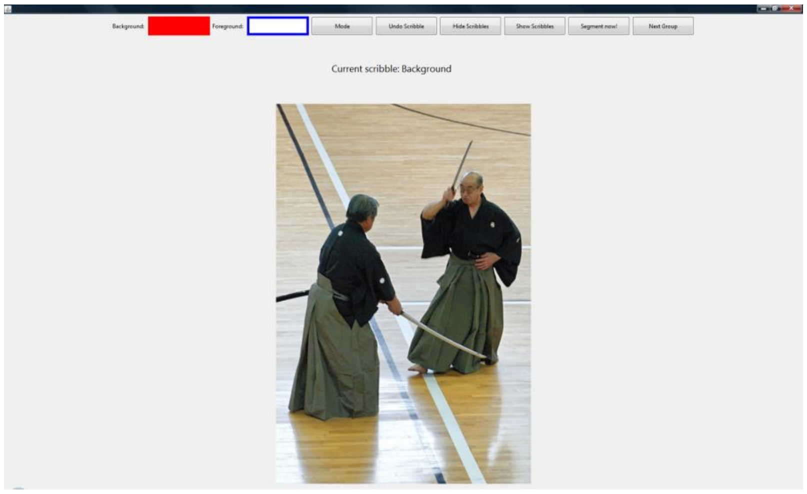


Figure 16: Interactive Co-segmentation GUI – Starting Screen

The starting screen shows all images in the first image set. At this point, the user selects an image on which to scribble, by clicking on that image. The GUI will then zoom in on that image.



Current scribble: Background

Figure 17: Interactive Co-segmentation GUI – Selected Image

There are several buttons on the top of the window. Below, ones needed to perform interactive co-segmentation are listed from left to right, along with their corresponding functions:

- Background/Foreground: These buttons control whether your current scribble should mark a foreground or background region in the image. If background scribbles are currently selected, the leftmost button will be solid red, and the "Current Scribble" dialog will change to say "Background". If foreground scribbles are currently selected, the second-to-leftmost button will be solid blue, and the "Current Scribble" dialog will change to say "Foreground".
- Mode: Switch between image selection and image scribbling screens.
- Undo Scribble: Undoes scribbles.
- Hide Scribbles: Hides all scribbles for the selected image.
- Show Scribbles: Shows all scribbles for the selected image.
- Segment now!: Performs co-segmentation with the current scribbles.
- Next Group: Switches to the next image group.

The image might look something like this after adding scribbles. Blue scribbles denote foreground and red scribbles denote background.

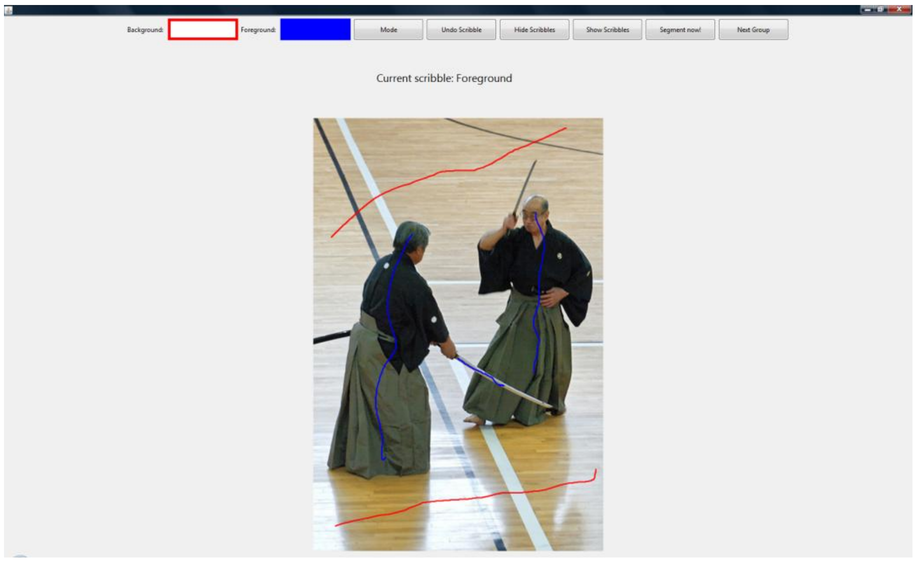
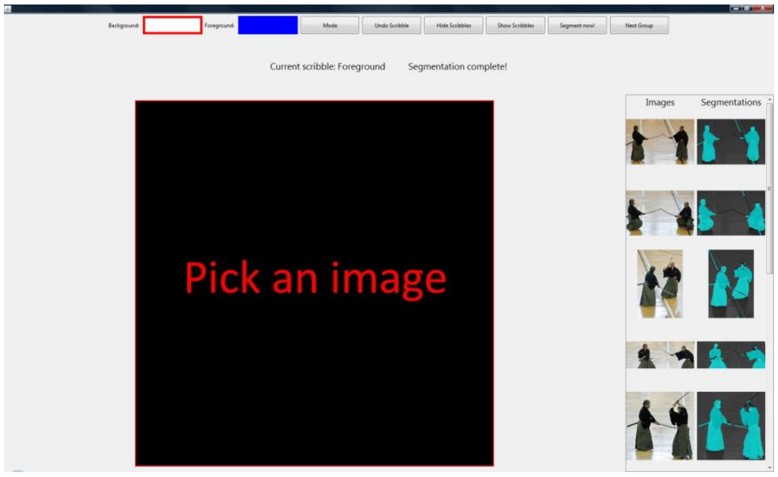


Figure 18: Interactive Co-segmentation GUI – After Adding Scribbles

Click "Segment now!" to begin co-segmentation. After the program is finished running, the screen will look like this:



Current scribble: Foreground Segmentation complete!

Figure 19: Interactive Co-segmentation GUI – After Co-segmentation

At this point, you can click on any image from the scrolling window on the right to enlarge it. You can also add scribbles to other images in the image set and re-run the co-segmentation algorithm, if you so choose.

Appendix B: Code

getColorModeLabels.m

This function assigns color mode labels to each pixel in the image using k-means clustering.

```
%Folder which contains all image folders to be processed
imgFolder = '';
%Folder which contains this file. Should also contain kmeans executable and all files
necessary to run it.
curFolder = '';

folderList = dir(imgFolder);
%Count starts at 3 since first two entries are always '.' And '..'
%Change end number based on number of image folders
for(h=3:40)

     folderGroup = folderList(h).name

    %Get name of each image in folder
    img_list = dir([imgFolder '/' folderGroup '/*.jpg']);

    start = clock;

    for(i=1:length(img_list))
        %Extract RGB data from image and put it into a file
        %Each row of the file contains RGB data for one pixel
        %Pixels are processed in row order
        imgName = img_list(i).name
        imgRGB = imread([imgFolder '/' folderGroup '/' imgName]);
        [height width numChannels] = size(imgRGB);

        %Use this to convert to HSV
        img = rgb2hsv(imgRGB);
        %Use this to convert to Luv
        img = RGB2Luv(imgRGB);
        %Use this if using RGB
        img = imgRGB;

        img = permute(img, [3 2 1]);
        img = reshape(img, [1 height*width*numChannels]);


        %Create .ds and .ds.universe files
        datFileName = imgName(1:end-4);
        fid = fopen([curFolder '/' datFileName '.ds'], 'w');
        fprintf(fid, 'x0 x1 x2\n');
        fprintf(fid, '%f %f %f\n', img);
        fclose(fid);

        fid = fopen([curFolder '/' datFileName '.ds.universe'], 'w');
        %Use these parameters for Luv image
        fprintf(fid, '0 100\n');
        fprintf(fid, '-134 220\n');
        fprintf(fid, '-140 122');
        %Use these parameters for HSV image
    %     fprintf(fid, '0 1\n');
    %     fprintf(fid, '0 1\n');
    %     fprintf(fid, '0 1');
        %Use these parameters for RGB image
    %     fprintf(fid, '0 255\n');
    %     fprintf(fid, '0 255\n');
    %     fprintf(fid, '0 255');
        fclose(fid);


        %Get color mode labels for current image file
        %ctrs - list of cluster centers
        %clust - groups image indices (row major order) by cluster (not
        %        necessarily in order listed in ctrs)
        %labels - assigns a label to each index between 0 and N-1, where N
        %         is the number of clusters
        input_filename = [datFileName '.ds'];
        ctrs_filename = ['ctrs_' datFileName];
        clust_filename = ['clust_' datFileName];
        labels_filename = ['labels_' datFileName];

        %Run xmeans to get pixel indices grouped by color mode
        system(['kmeans kmeans -k 1 -method blacklist -max_leaf_size 40 ' ...
                '-min_box_width 0.03 -cutoff_factor 0.5 -max_iter 200 ' ...
                '-num_splits 6 -max_ctrs 15 -in ' input_filename ...
                ' -save_ctrs ' ctrs_filename ' -printclusters ' clust_filename]);
        system(['kmeans membership -in ' clust_filename ' >' labels_filename]);

        %labels_filename contains pixel indices grouped by color mode, color modes
        separated by blank line
        %Assign label to each pixel
        colorModeLabel = 0;
        colorModeLabels = zeros(1, height*width);
        fid = fopen([curFolder '/' labels_filename], 'r');
        while(1)
            curIdx = fgets(fid);
            if(curIdx==-1)      %EOF
                break;
            elseif(isempty(str2num(curIdx)))    %blank space line
                colorModeLabel = colorModeLabel + 1;
            else
                colorModeLabels(str2num(curIdx)+1) = colorModeLabel;
            end
        end
        fclose(fid);

        %Write labels to a file
        fid = fopen([imgFolder '/' folderGroup '/modes_' datFileName], 'w');
        fprintf(fid, '%d\n', colorModeLabels);
        fclose(fid);
    end

end
```

RGB2Luv.m

Converts an RGB image to Luv color space

```
function luvim = RGB2Luv(im)

%Ensure input image has 3 color channels
if(size(im,3) ~= 3)
    error('im must have three color channels');
end
if(max(im(:)) > 1)
    im = im./255;
end


XYZ = [.4125 .3576 .1804; .2125 .7154 .0721; .0193 .1192 .9502];
Yn = 1.0;
Lt = .008856;
Up = 0.19784977571475;
Vp = 0.46834507665248;
imsiz = size(im);
im = permute(im,[3 1 2]);
im = double(reshape(im,[3 imsiz(1)*imsiz(2)]));
xyz = reshape((XYZ*im)',imsiz);
x = xyz(:,:,1);
y = xyz(:,:,2);
z = xyz(:,:,3);

l0 = y./Yn;
l = l0;
l(l0>Lt) = 116.*(l0(l0>Lt).^(1/3)) - 16;
l(l0<=Lt) = 903.3*l0(l0<=Lt);
c = x + 15*y + 3 * z;
u = 4*ones(imsiz(1:2),class(im));
v = (9/15)*ones(imsiz(1:2),class(im));
u(c~=0) = 4*x(c~=0)./c(c~=0);
v(c~=0) = 9*y(c~=0)./c(c~=0);

u = 13*l.*(u-Up);
v = 13*l.*(v-Vp);

luvim = cat(3,l,u,v);
```

lcpExtractHSV_pix.cpp

This MEX-function extracts the LCP feature from an HSV image. It can also be used to extract LCP features from an Luv image.

```
#include <stdio.h>
#include "mex.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]);
void getLCP(int height, int width, int *labels, double *imgHSV, double *lcpOutput);
int findColorModeIdxNorth(int i, int j, int height, int width, int *colorModes);
int findColorModeIdxEast(int i, int j, int height, int width, int *colorModes);
int findColorModeIdxSouth(int i, int j, int height, int width, int *colorModes);
int findColorModeIdxWest(int i, int j, int height, int width, int *colorModes);


//Inputs: height, width - dimensions of image
//     labels - length (height*width) array containing color mode label data
//          entries are row major order
//     imgHSV - length (3*height*width) array containing HSV image data
//          entries are row major order
//Outputs:   lcpOutput - length (height*width*15) array containing HSV values of
//             each pixel as well as nearest pixels in NESW directions
//             that belong to a different color mode
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
   int *labels;
   double *imgHSV;
   int height, width;
   double *lcpOutput;

   //Get height and width of original image
   height = mxGetScalar(prhs[0]);
   width = mxGetScalar(prhs[1]);
   //Get labels for each pixel
   labels = (int *)(mxGetPr(prhs[2]));
   //Get HSV colors for each pixel
   imgHSV = (double *)(mxGetPr(prhs[3]));


   //Output mex array
   plhs[0] = mxCreateDoubleMatrix(1, height*width*15, mxREAL);
   lcpOutput = (double *)(mxGetPr(plhs[0]));

   //Obtain local color pattern of image
   getLCP(height, width, labels, imgHSV, lcpOutput);
}


void getLCP(int height, int width, int *labels, double *imgHSV, double *lcpOutput)
{
   int i, j;
   //Distance from current point to nearest color mode in 4 cardinal directions
   int colorModeIdxN, colorModeIdxE, colorModeIdxS, colorModeIdxW, colorModeIdxC;

   for(i=0; i<height; i++)
   {
      for(j=0; j<width; j++)
      {
         //Locate nearest color mode boundaries in 4 cardinal directions
```

```
            //Assume edges of image are always color mode boundaries

            colorModeIdxN = findColorModeIdxNorth(i,j, height, width, labels);
            colorModeIdxE = findColorModeIdxEast(i,j, height, width, labels);
            colorModeIdxS = findColorModeIdxSouth(i,j, height, width, labels);
            colorModeIdxW = findColorModeIdxWest(i,j, height, width, labels);
            colorModeIdxC = width*i+j;

            lcpOutput[i*15*width + j*15] = imgHSV[colorModeIdxC*3];
            lcpOutput[i*15*width + j*15 + 1] = imgHSV[colorModeIdxC*3 + 1];
            lcpOutput[i*15*width + j*15 + 2] = imgHSV[colorModeIdxC*3 + 2];
            lcpOutput[i*15*width + j*15 + 3] = imgHSV[colorModeIdxN*3];
            lcpOutput[i*15*width + j*15 + 4] = imgHSV[colorModeIdxN*3 + 1];
            lcpOutput[i*15*width + j*15 + 5] = imgHSV[colorModeIdxN*3 + 2];
            lcpOutput[i*15*width + j*15 + 6] = imgHSV[colorModeIdxE*3];
            lcpOutput[i*15*width + j*15 + 7] = imgHSV[colorModeIdxE*3 + 1];
            lcpOutput[i*15*width + j*15 + 8] = imgHSV[colorModeIdxE*3 + 2];
            lcpOutput[i*15*width + j*15 + 9] = imgHSV[colorModeIdxS*3];
            lcpOutput[i*15*width + j*15 + 10] = imgHSV[colorModeIdxS*3 + 1];
            lcpOutput[i*15*width + j*15 + 11] = imgHSV[colorModeIdxS*3 + 2];
            lcpOutput[i*15*width + j*15 + 12] = imgHSV[colorModeIdxW*3];
            lcpOutput[i*15*width + j*15 + 13] = imgHSV[colorModeIdxW*3 + 1];
            lcpOutput[i*15*width + j*15 + 14] = imgHSV[colorModeIdxW*3 + 2];
        }
    }
}


int findColorModeIdxNorth(int i, int j, int height, int width, int *colorModes)
{
    int temp = i;

    //Stop moving up once we hit the top row or a different color mode
    while(temp>0 && colorModes[width*i+j]==colorModes[width*temp+j])
    {
        temp--;
    }

    return width*temp+j;
}

int findColorModeIdxEast(int i, int j, int height, int width, int *colorModes)
{
    int temp = j;

    //Stop moving up once we hit the rightmost column or a different color mode
    while(temp<width-1 && colorModes[width*i+j]==colorModes[width*i+temp])
    {
        temp++;
    }

    return width*i+temp;
}

int findColorModeIdxSouth(int i, int j, int height, int width, int *colorModes)
{
    int temp = i;

    //Stop moving up once we hit the top row or a different color mode
    while(temp<height-1 && colorModes[width*i+j]==colorModes[width*temp+j])
    {
        temp++;
    }
    return width*temp+j;
}

int findColorModeIdxWest(int i, int j, int height, int width, int *colorModes)
{
    int temp = j;

    //Stop moving up once we hit the rightmost column or a different color mode
    while(temp>0 && colorModes[width*i+j]==colorModes[width*i+temp])
    {
        temp--;
    }

    return width*i+temp;
}
```

lcpExtractRGB_pix.cpp

This MEX-function extracts the LCP feature from an RGB image. It is pretty much exactly the same as lcpExtractHSV_pix.cpp, except that the function requires integer inputs.

```
#include <stdio.h>
#include "mex.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]);
void getLCP(int height, int width, int *labels, int *imgRGB, double *lcpOutput);
int findColorModeIdxNorth(int i, int j, int height, int width, int *colorModes);
int findColorModeIdxEast(int i, int j, int height, int width, int *colorModes);
int findColorModeIdxSouth(int i, int j, int height, int width, int *colorModes);
int findColorModeIdxWest(int i, int j, int height, int width, int *colorModes);


//Inputs: height, width - dimensions of image
//     labels - length (height*width) array containing color mode label data
//          entries are row major order
//     imgRGB - length (3*height*width) array containing RGB image data
//               entries are row major order
//Outputs:   lcpOutput - length (height*width*15) array containing RGB values of
//              each pixel as well as nearest pixels in NESW directions
//              that belong to a different color mode
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
   int *labels;
   int *imgRGB;
   int height, width;
   double *lcpOutput;

   //Get height and width of original image
   height = mxGetScalar(prhs[0]);
   width = mxGetScalar(prhs[1]);
   //Get labels for each pixel
   labels = (int *)(mxGetPr(prhs[2]));
   //Get RGB colors for each pixel
   imgRGB = (int *)(mxGetPr(prhs[3]));


   //Output mex array
   plhs[0] = mxCreateDoubleMatrix(1, height*width*15, mxREAL);
   lcpOutput = (double *)(mxGetPr(plhs[0]));

   //Obtain local color pattern of image
   getLCP(height, width, labels, imgRGB, lcpOutput);
}


void getLCP(int height, int width, int *labels, int *imgRGB, double *lcpOutput)
{
   int i, j;
   //Distance from current point to nearest color mode in 4 cardinal directions
   int colorModeIdxN, colorModeIdxE, colorModeIdxS, colorModeIdxW, colorModeIdxC;

   for(i=0; i<height; i++)
   {
      for(j=0; j<width; j++)
      {
         //Locate nearest color mode boundaries in 4 cardinal directions
```

```
            //Assume edges of image are always color mode boundaries

            colorModeIdxN = findColorModeIdxNorth(i,j, height, width, labels);
            colorModeIdxE = findColorModeIdxEast(i,j, height, width, labels);
            colorModeIdxS = findColorModeIdxSouth(i,j, height, width, labels);
            colorModeIdxW = findColorModeIdxWest(i,j, height, width, labels);
            colorModeIdxC = width*i+j;

            lcpOutput[i*15*width + j*15] = imgRGB[colorModeIdxC*3];
            lcpOutput[i*15*width + j*15 + 1] = imgRGB[colorModeIdxC*3 + 1];
            lcpOutput[i*15*width + j*15 + 2] = imgRGB[colorModeIdxC*3 + 2];
            lcpOutput[i*15*width + j*15 + 3] = imgRGB[colorModeIdxN*3];
            lcpOutput[i*15*width + j*15 + 4] = imgRGB[colorModeIdxN*3 + 1];
            lcpOutput[i*15*width + j*15 + 5] = imgRGB[colorModeIdxN*3 + 2];
            lcpOutput[i*15*width + j*15 + 6] = imgRGB[colorModeIdxE*3];
            lcpOutput[i*15*width + j*15 + 7] = imgRGB[colorModeIdxE*3 + 1];
            lcpOutput[i*15*width + j*15 + 8] = imgRGB[colorModeIdxE*3 + 2];
            lcpOutput[i*15*width + j*15 + 9] = imgRGB[colorModeIdxS*3];
            lcpOutput[i*15*width + j*15 + 10] = imgRGB[colorModeIdxS*3 + 1];
            lcpOutput[i*15*width + j*15 + 11] = imgRGB[colorModeIdxS*3 + 2];
            lcpOutput[i*15*width + j*15 + 12] = imgRGB[colorModeIdxW*3];
            lcpOutput[i*15*width + j*15 + 13] = imgRGB[colorModeIdxW*3 + 1];
            lcpOutput[i*15*width + j*15 + 14] = imgRGB[colorModeIdxW*3 + 2];
        }
    }
}


int findColorModeIdxNorth(int i, int j, int height, int width, int *colorModes)
{
    int temp = i;

    //Stop moving up once we hit the top row or a different color mode
    while(temp>0 && colorModes[width*i+j]==colorModes[width*temp+j])
    {
        temp--;
    }

    return width*temp+j;
}

int findColorModeIdxEast(int i, int j, int height, int width, int *colorModes)
{
    int temp = j;

    //Stop moving up once we hit the rightmost column or a different color mode
    while(temp<width-1 && colorModes[width*i+j]==colorModes[width*i+temp])
    {
        temp++;
    }

    return width*i+temp;
}

int findColorModeIdxSouth(int i, int j, int height, int width, int *colorModes)
{
    int temp = i;

    //Stop moving up once we hit the top row or a different color mode
    while(temp<height-1 && colorModes[width*i+j]==colorModes[width*temp+j])
    {
        temp++;
    }
    return width*temp+j;
}

int findColorModeIdxWest(int i, int j, int height, int width, int *colorModes)
{
    int temp = j;

    //Stop moving up once we hit the rightmost column or a different color mode
    while(temp>0 && colorModes[width*i+j]==colorModes[width*i+temp])
    {
        temp--;
    }

    return width*i+temp;
}
```