Image Processing for Video-CT Registration in Sinus Surgery

John Lee | Kyle Xiong | Calvin Zhao

Computer Integrated Surgery II

Final Report

Background Information

Currently, magnetic trackers are common in robotic microsurgery where maintaining sight is difficult. Magnetic trackers are relatively inexpensive and do not require a line of sight to know where the tool is. However, they easily interfere with instruments, especially metal ones. High resolution magnetic trackers are also expensive while cheaper ones are generally inaccurate. Therefore to remedy the general inaccuracy of this technology we want to correlate high resolution CT scans of the patient with video feed from the endoscope. By registering the contours from video data to the CT image, we will be able to track the position of the endoscope.

Our project aims to fulfill the previously stated objective. To this end, we have designed and implemented an algorithm to automatically extract occluding contours from sinus surgery videos. Common edge detection algorithms such as the Sobel, Canny and Roberts edge detection methods do not work well for tissue edge detection because they detect both boundary edges (true edges) and texture edges. Our algorithm is able to determine which edges are true positives and false positives and provides an accurate representation of where occluding contours are in the video frames. The algorithm returns the edge image derived from an input image, the normal slopes of each edge pixel, and the x and y coordinates of each normal vector.

The project will eventually be used with an existing algorithm that registers our edge and normal data to CT image data. By doing so we can establish a link between video from surgical endoscopes to CT images and determine the position of the endoscope in the context of the CT image. This helps surgeons track endoscope position during surgery and can potentially increase the accuracy of magnetic tracking. In the end, the sinus surgery project can be used to develop an augmented reality program which delineates anatomical features during surgery. We hope that our algorithm, currently implemented in MATLAB, can be further optimized and implemented in C/C++ in the future. Ideally we want detection to run in 50 - 100 milliseconds so surgeons get close to real-time feedback.

Approach and Methods

Occluding contours are notably different than edges since they define the boundary of physical objects. This complicates detection, as some high contrast edges may be results of texture and not geometry. Our detection algorithm will be written, tested and optimized in MATLAB. However, in order to perform real-time contour detection, we will have to implement our code in C/C++ in the future. We have examined various existing methods of contour detection and eliminated most of these methods because they would not produce desirable results.

First and foremost, we used single-image edge detection methods. Single-image contour detection has been attempted by various groups over the years. One of the most basic and famous algorithms is the Canny edge detector, developed by John Canny in 1986. This algorithm examines contrast within the image to find edges and applies a series of filters to remove noise. However this method ultimately did not distinguish between textured and occluding edges, as shown in Figure 1.

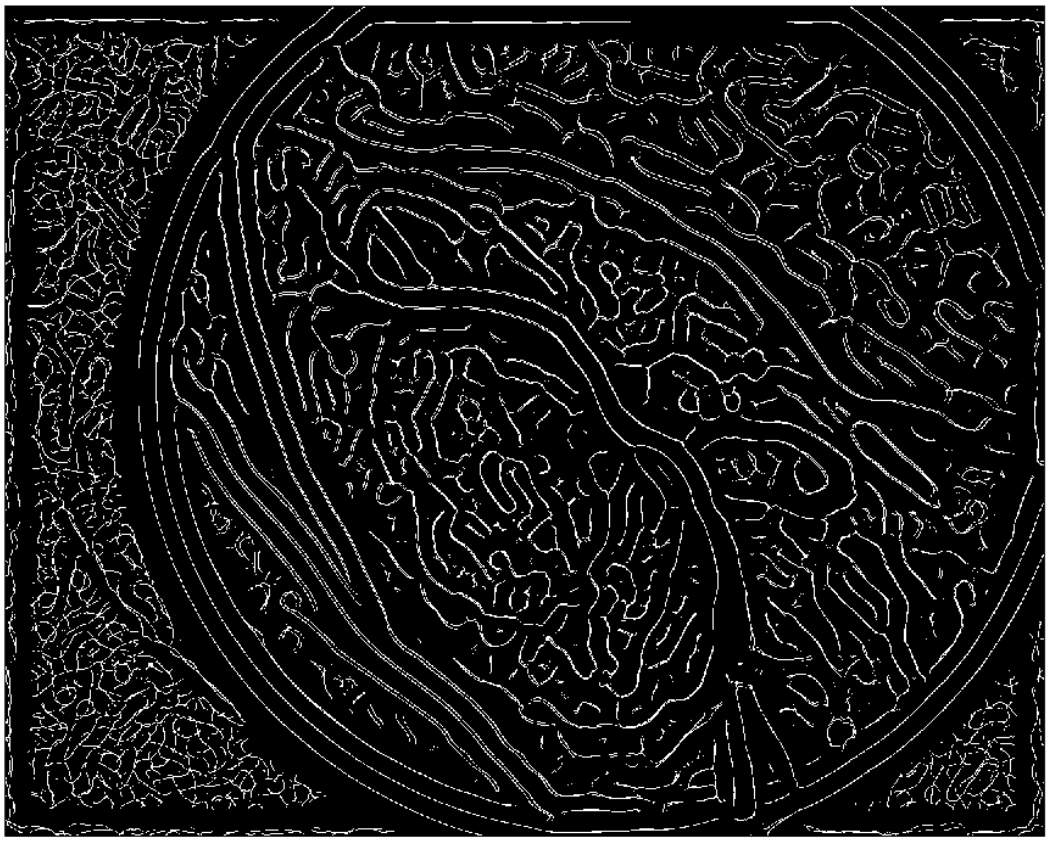


FIGURE 1. RESULT OF EDGE DETECTION USING CANNY'S METHOD.

Single-image contour detection also does not consider temporal data. We believed that utilizing multiple video frames can allow for more accurate occluding contour detection. For this, we used optical flow to detect the motion of occluding contours across multiple frames. Optical flow is a well-established computer vision method that tracks points across sequential

images. These points are distinct physical features within the image that shift as the camera or feature moves. If enough of the image is sampled, a vector displacement map can be constructed that shows the movement of features in the video. Since objects closer to the camera behave differently than objects farther away, namely they have greater displacement, the optical flow algorithm will allow us to discern edges by examining regions of similar movement.

While optical flow sounded great on paper, it produced undesirable results as well. As seen in Figure 2, the optical flow algorithm returned large magnitude motion vectors for pixels that weren't on occluding contours. This was due to the algorithm detecting motion in the texture of the tissue. To address this we tried Gaussian blurring of the whole image but blurring couldn't fully eliminate the color contrast on the tissue surface and motion vectors still appeared where they shouldn't have (also shown in Figure 2). Therefore we had to rule out using only optical flow to differentiate between texture and boundary edges.

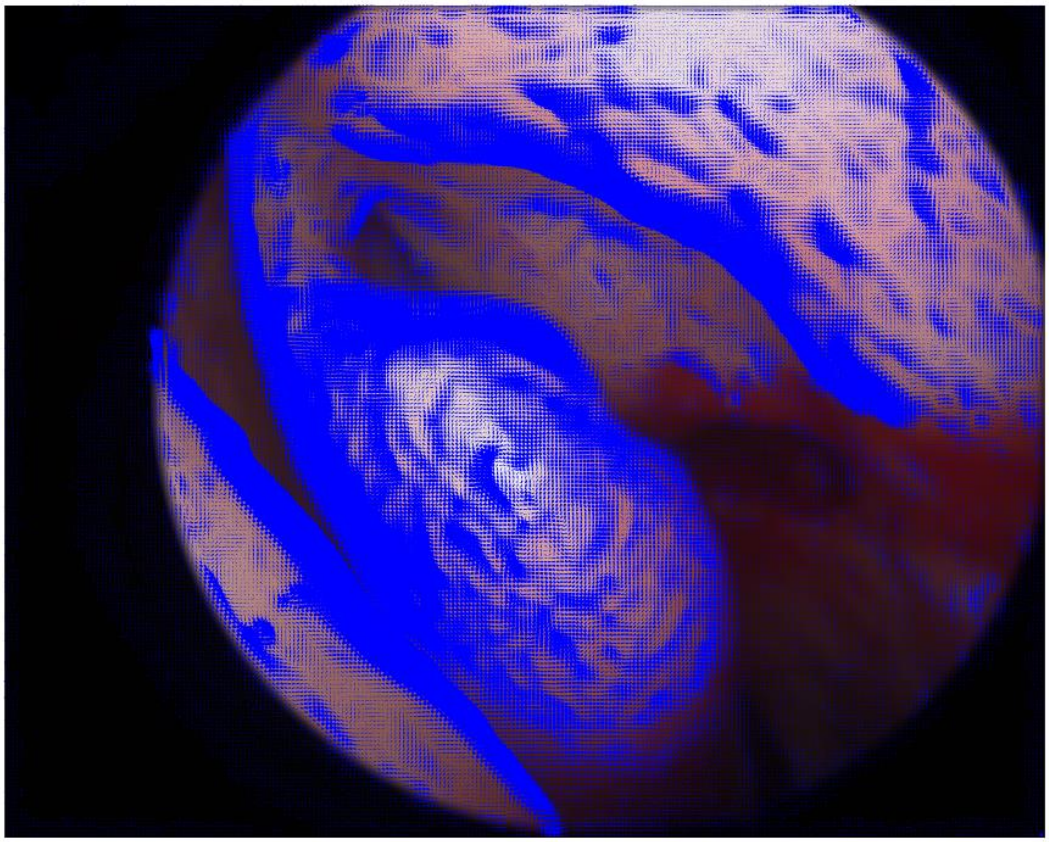


FIGURE 2. RESULT OF USING OPTICAL FLOW TO DETECT OCCLUDING CONTOURS AFTER HEAVY GAUSSIAN BLURRING.

Since neither of the aforementioned methods yielded favorable results we hypothesized that if we used the methods simultaneously we may get a better looking edge image. To this effect we ran both single-image edge detection and ran optical flow for multiple frames and got the noisy edge detection image and a matrix of motion vectors. With the motion vectors we

calculated the uniformity of the motion vectors as the variance of a neighborhood of pixels around each pixel in the image. We applied a smoothness filter to ignore pixels with large variance (low smoothness) and an intensity filter to ignore pixels that were too bright (filter shown in Figure 3). We then applied a logical AND function to the filter and the edge detection image, effectively removing noisy pixels and very bright pixels. The results were satisfactory (Figure 3) but some problems with the method still persisted. There were too many parameters that could not be dynamically modified. As a result this method failed with many of our frames and likely couldn't be used universally. In addition, combining the methods didn't solve the question of how to eliminate texture edges altogether. Up to this point the method still doesn't address conclusively how to differentiate between texture and boundary edges.

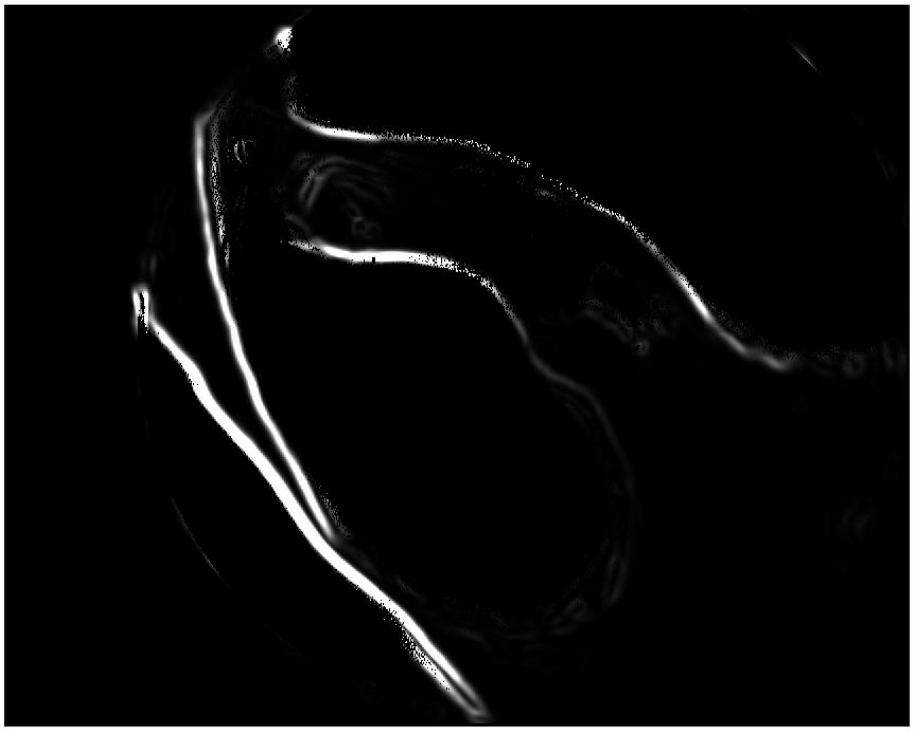
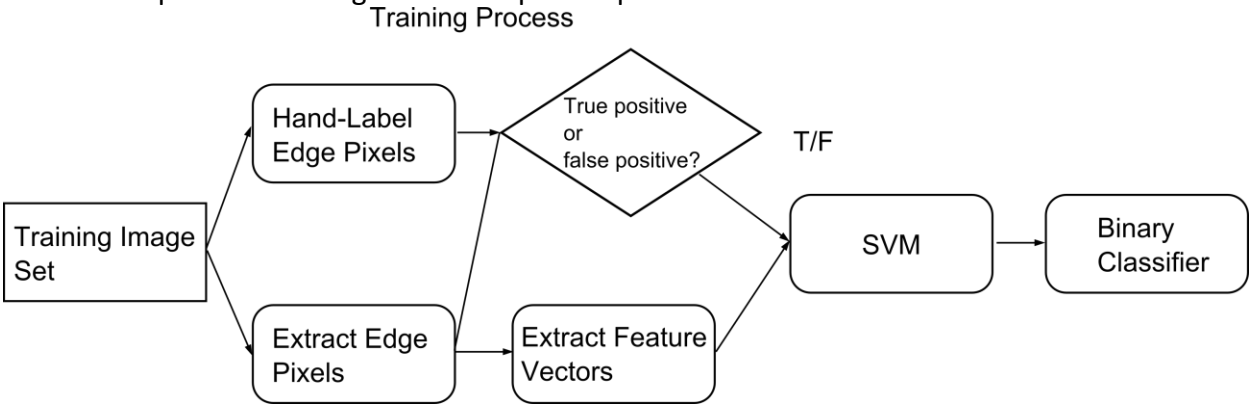


FIGURE 3. TOP: SMOOTHNESS AND INTENSITY FILTER. BOTTOM: RESULT OF INTEGRATING SMOOTHNESS/INTENSITY FILTER WITH FIGURE 2'S EDGE DETECTION IMAGE.

We thus decided to start over with algorithm implementation and train support vector machines (SVM) to help us consistently classify texture edges and boundary edges. Because our surgical video frames are relatively uniform in shape, texture and lighting, we believed we can train an SVM and construct a robust classifier that allows for accurate classification of true positives and false positives. Figure 4 shows the workflow of training the SVM and using the classifier to predict true edges in subsequent inputs.



Prediction

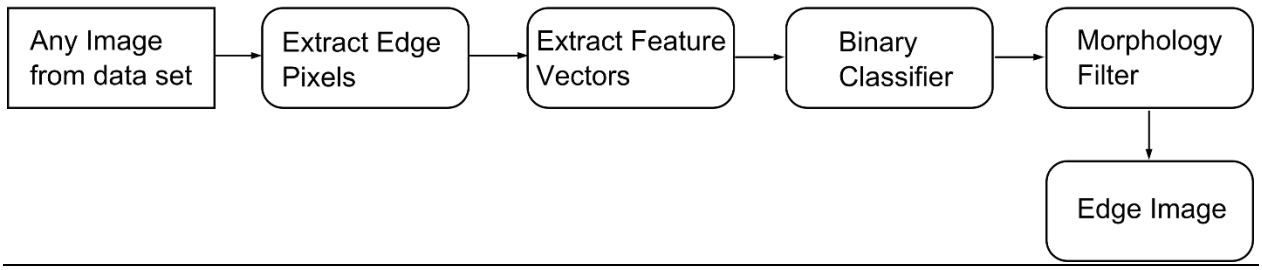


FIGURE 4. TOP: FLOW DIAGRAM OF THE SVM TRAINING PROCESS. BOTTOM: FLOW DIAGRAM OF THE TRUE EDGE PREDICTION PROCESS.

To train the SVM we used a set of ten hand-labeled images as inputs, selected from the initial data set we were given. The MATLAB program finds and saves the position of pixels that are labeled in the image. The images are then filtered out by their red channels and amplified to accentuate the contrast. The program then applies a light Gaussian blur so edge detection doesn't provide too many points. Afterwards, the program divides the images into two vectors: the edge feature vector and non-edge feature vector. Edge features are features corresponding to pixels in the edge detection image that are on true boundary edges while non-edge features correspond to pixels that are on texture edges. After edge and non-edge features are extracted and separately saved, the two vectors are combined into one vector and a label vector is created that labels each feature as either "1" (true positive) or "0" (false positive). The feature vector and the label vector are then put into MATLAB's fitcsvm() function for training. The result is a classification variable "svm" that we will use for prediction.

The features we used to classify true edges and non-edges are:

- Local histogram of image intensity
- Gradients of scaled down images
- Local edge strength from histograms' chi-squared distance

We believe that by using these three features the learning algorithm will have enough information to differentiate between textures and boundary edges. The local histogram provides the distribution of image intensity values in a circular region around each pixel. The gradients of scaled down images serve to remove textures and enhance contour edges. The local edge strength was a custom algorithm created from taking a disk around a pixel in a specified radius and bifurcating the disk at various angles. Histograms from each half of the disk was calculated as well as the chi-squared distance. The maximum chi squared distance out of all the angles sampled was deemed the local edge strength. This algorithm worked very well assuming edges were locally straight and featured different contrasts on either side.

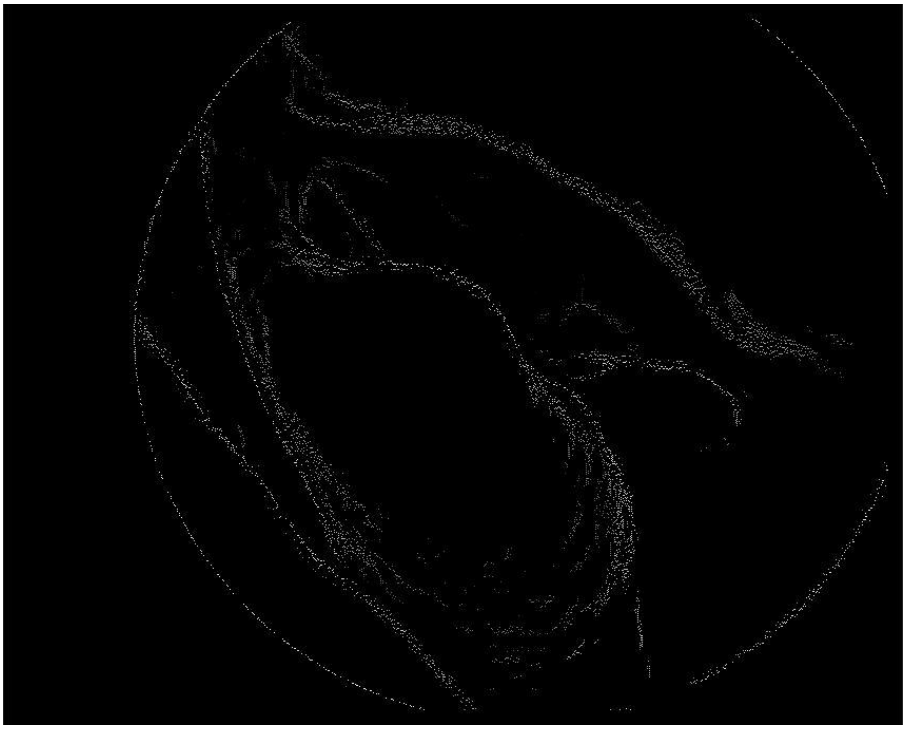


FIGURE 5. LOCAL EDGE STRENGTH RESULTS. WHITE INTENSITY CORRESPONDS TO HIGHER EDGE STRENGTH

Predicting true edges in any image required using the previously defined classification variable. We chose any image as an input and extracted all features from every point in the edge detection, saved in a variable “features”. Using MATLAB’s predict() function with “svm” and “features” as its inputs, the program attempted to predict true positives. Once the predicted edges were saved by MATLAB, our program then computed the normal slopes. We used the polyfit() function in MATLAB to fit a first degree polynomial to a cluster of pixels around each predicted edge pixel and let the normal slope of that pixel be the perpendicular to the polynomial’s slope. This gave a reasonable approximation of normals for each point. Finally, the program saves the corresponding x and y coordinates of each normal slope.

The MATLAB program takes all images in the input directory with the naming scheme “rec-000098##” as its inputs, although the scheme, directory and input mechanism can be easily changed in the source code.

After completing the SVM training, our program’s outputs are:

- The SVM classification variable
- Non-edge features detection image
- Matrix with all hand-labeled contours

The final outputs of the prediction process are:

- Contour detection image
- Normal vector slope matrix
- X and Y coordinates of normal vectors

We initially used Roberts' edge detection method in SVM training and prediction. Figure 5 shows that by using this method we were able to achieve an edge detection image with almost no false positives. However, the main problem with using Roberts' was that all detected points were not on a continuous line, which we needed for both the registration algorithm to work and to find normal vectors for each edge pixel.

FIGURE 6. SIDE-BY-SIDE COMPARISON OF TRUE EDGES AND NON-EDGES AFTER USING ROBERTS' EDGE DETECTION IN SVM TRAINING.

To acquire a one-pixel thick line for the registration algorithm, we applied morphological filters to the image on the left in Figure 5. We began by dilating the image using MATLAB's imdilate() function and a disk of radius 5 as the structuring element. We applied the dilation twice and then used MATLAB's bwmorph() function and passed 'thin' and 'inf' as parameters to thin the dilated image and produce continuous lines. However, as shown in Figure 6, applying the morphological filters generated unwanted branches and could be seen as a compromise in contour extraction accuracy. Therefore, to remedy this we decided to take an alternate approach, one that will lead us to our current results.

FIGURE 7. EDGE DETECTION IMAGE AFTER APPLYING TWO DILATIONS AND A THINNING MORPHOLOGICAL OPERATION ON THE ROBERTS-BASED PREDICTION.

Results

Our previous hand-labeled ground truths, shown in Figure 7-Left, were drawn arbitrarily and the labeled lines were too thick and could have included some textures, affecting the SVM's decisions during the prediction phase. To get an absolute ground truth with a line-width of 1, we decided to switch from Roberts to Canny edge detection. The hand-labeled ground truths in Figure 7-Right were generated by first taking a full Canny edge detection image and removing the lines that weren't occluding contours.

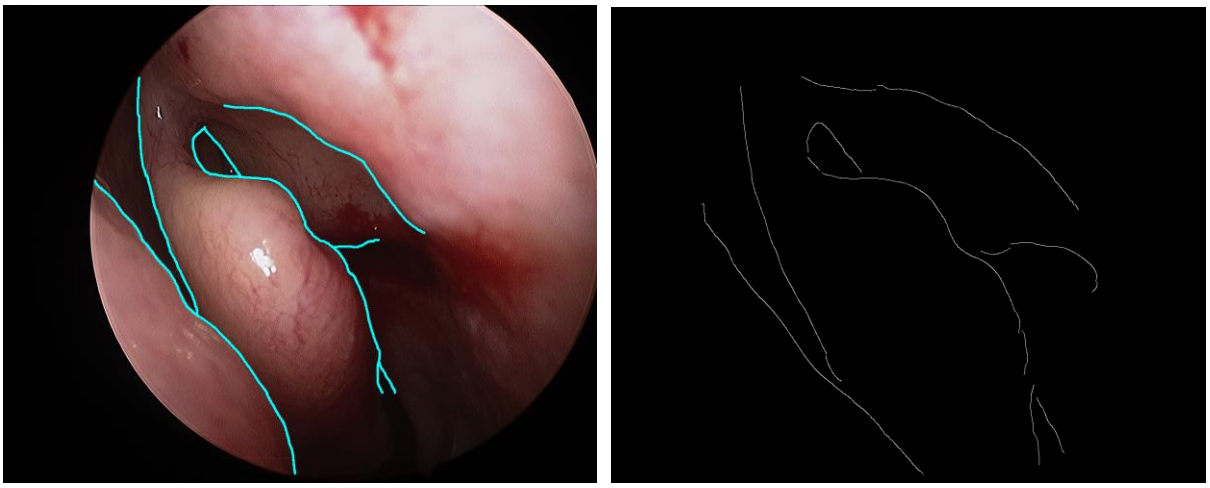


FIGURE 8. LEFT: HAND-LABELED IMAGE USED FOR TRAINING IN ROBERTS-BASED SVM CLASSIFICATION. RIGHT: AN ADJUSTMENT OF OUR HAND-LABELING STANDARDS IN CANNY-BASED SVM CLASSIFICATION.

When Canny-based hand-labeled was finished, we applied Canny to the rest of the training and prediction phases, and our new results are shown in Figure 8. After seeing the initial results, we realized that the image had a lot of noise and we applied less taxing morphological filters to remove them. We used MATLAB's imclose() function using a disk of radius 3 as the structuring element, and MATLAB's bwareaopen() function using a threshold of 15. Thus the program attempted to connect nearby disconnected pixels and removed any remaining pixel clusters that had fewer than 15 pixels.

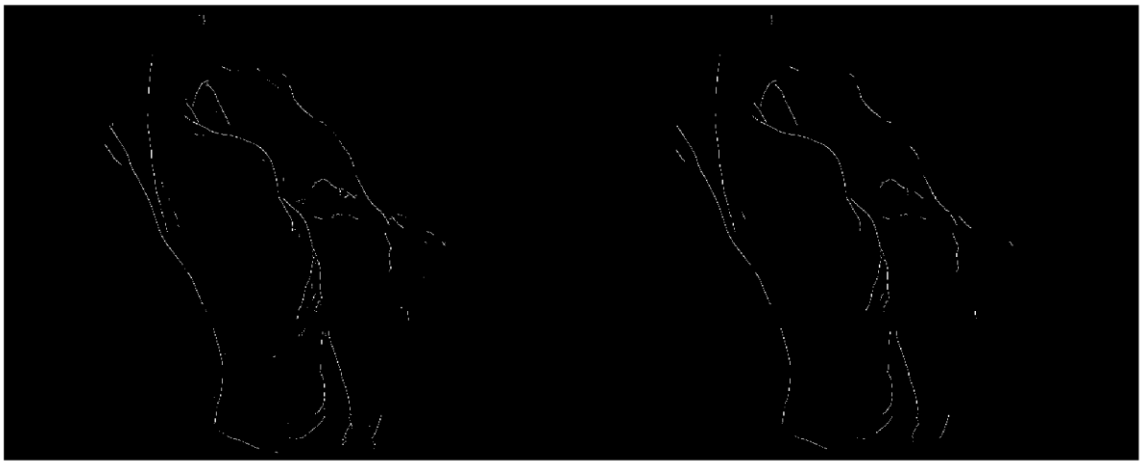


FIGURE 9. LEFT: EDGE DETECTION AFTER USING CANNY FOR PREDICTION. RIGHT: EDGE DETECTION IMAGE AFTER APPLYING CLOSING AND OPENING TO THE LEFT IMAGE.

Although we have a rudimentary normal calculation function, it is not yet ideal and as of now not easy to visualize. Figure 9 therefore only depicts sections of the image in which we see that the normal vectors appear to have the correct slope in the context of the surrounding pixels.

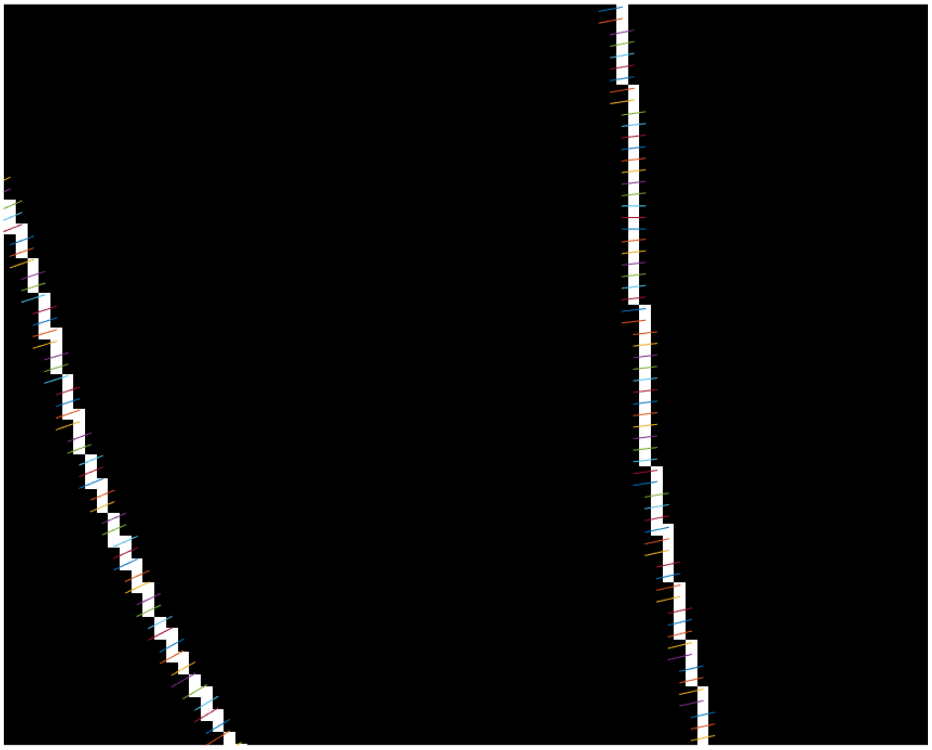


FIGURE 10. NORMAL SLOPES (COLORED LINES) PLOTTED ON THE EDGE IMAGE, CALCULATED FROM FITTING A NEIGHBORHOOD OF PIXELS TO A POLYNOMIAL FUNCTION.

Significance

Currently, magnetic trackers are common in robotic microsurgery where maintaining sight is difficult. Magnetic trackers are relatively inexpensive and do not require a line of sight to know where the tool is. However, they easily interfere with instruments, especially metal ones. High resolution magnetic trackers are also expensive while cheaper ones are generally inaccurate. Our contours can be used with a separate registration algorithm that will allow surgeons to track the location of their instruments with high resolution preoperative CT image data.

Management Summary

Contributions

John created the framework for SVM training and developed the scaled gradient feature. Kyle developed the intensity histogram metric, and increased efficiency of feature extraction programs. Calvin developed and optimized the local edge strength chi squared distance metric

Accomplished vs. Planned

From our initial timeline, we were planning on creating the contour detection algorithm halfway through the semester, then work on interfacing with the CT registration algorithm and hopefully work on an augmented reality interface. We were going to use a combination of simple edge detection (e.g. Canny) combined with optical flow information between frames to determine the location of contours. However, due to complications with optical flow, we were forced to find an alternative method of completing our algorithm. We had to dedicate the rest of the semester working on building out the alternative method, which used an SVM classifier to classify edges as true contours and false contours. Developing this alternate method meant that we did not have time to work on interfacing our algorithm with the registration algorithm or any of the augmented reality functionality.

Future Development

The next steps for development would be to finish the next steps in our initial timeline. First, we would need to interface our contour detection algorithm with the CT registration algorithm. Once the combined algorithms is able to register occluding contours from video data with CT data, the next step would be to create a method of tracking the endoscope in real time. This can be accomplished by first reimplementing our code in a faster language, like C, to make sure that it can detect and predict features on the order of milliseconds instead of minutes. Next, the tracking method must be able to consistently register the video data to the CT data so that the system knows where the endoscope is in terms of CT coordinates. Finally, after a tracking and registration method is completed, an augmented reality overlay could be developed. This AR overlay would use the tracking information to point out important features, such as nerves and other important tissue, to surgeons intraoperatively.

Lessons Learned

This project taught us a lot about the basics of computer vision techniques and some of their applications. We learned that although optical flow seemed promising for preserving temporal information throughout video frames, it was not able to accurately find occluding contours. Although we were never able to completely identify why, we suspected that it was finding discontinuities in flow in texture edges in addition to contour edges. This made optical flow all but useless as a metric for finding contour edges.

Once we started using SVM, we learned a lot of which features were useful for finding contour edges. One of the lessons we learned in our search for features were that we needed to have large enough kernel/window sizes so that texture edges would not erroneously be picked up as contours in the SVM classifier. As long as we had large enough windows for our features, texture edges would be mostly be classified as noise.

Publications

- "Determining Optical Flow", Proc. SPIE 0281, Techniques and Applications of Image Understanding, 319 (November 12, 1981); doi:10.1117/12.965761
- “Contour Detection and Hierarchical Image Segmentation”, Arbelaez P., Fowlkes C., IEEE, Pattern Analysis and Machine Intelligence, Vol. 33, Issue 5 (March 22, 2011)
- Z Li, Y Liu, J Yun, F Huang, “A New Edge Detection Method Based on Contrast Enhancement,” IEEE, December 2009.
- D. Burschka et al., “Scale-invariant registration of monocular endoscopic images to CT-scans for sinus surgery”

Technical Appendix

Contents

Drivers

```
% gifOoutput.m
% Creates an animated gif of sequential frames

imageHeader = 'rec-000098';
imageType = '.bmp';
outputHeader = fullfile('output');
```

```
edgeHeader = fullfile(outputHeader, 'edges');
smoothHeader = fullfile(outputHeader, 'smoothness');
finalHeader = fullfile(outputHeader, 'final');

edgeGif = fullfile(outputHeader, 'edges.gif');
smoothGif = fullfile(outputHeader, 'smoothness.gif');
finalGif = fullfile(outputHeader, 'final.gif');

delay = 0.3;

for i = 20:68
    iStr = num2str(i);
    iNextStr = num2str(i+1);
    I1 = imread(fullfile(edgeHeader, strcat(imageHeader, iStr,'-
',iNextStr,imageType)));
    [A1, map1] = gray2ind(I1, 256);
    I2 = imread(fullfile(smoothHeader, strcat(imageHeader, iStr,'-
',iNextStr,imageType)));
    [A2, map2] = gray2ind(I2, 256);
    I3 = imread(fullfile(finalHeader, strcat(imageHeader, iStr,'-
',iNextStr,imageType)));
    [A3, map3] = rgb2ind(I3, 256);

    if i == 20
        imwrite(A1,map1,edgeGif,'gif','LoopCount',Inf,'DelayTime',delay);
        imwrite(A2,map2,smoothGif,'gif','LoopCount',Inf,'DelayTime',delay);
        imwrite(A3,map3,finalGif,'gif','LoopCount',Inf,'DelayTime',delay);
    else

imwrite(A1,map1,edgeGif,'gif','WriteMode','append','DelayTime',delay);

imwrite(A2,map2,smoothGif,'gif','WriteMode','append','DelayTime',delay);

imwrite(A3,map3,finalGif,'gif','WriteMode','append','DelayTime',delay);

    end
end
```

```
% labelDriver.m
% Exports basic edge detection images

close;
imageHeader = 'rec-000098';
imageType = '.bmp';
labeledImageType = '.png';
outputHeader = fullfile('output');
inputHeader = fullfile('input');
image = '20';

I1 = imread(fullfile('input', strcat(imageHeader, image, imageType)));
filter = fspecial('gaussian',20,4);
I1 = imfilter(I1,filter);
red = I1(:,:,1);
```

```
red_highC = imadjust(red,[0 0.7],[0 1]);

thresh = .000;

edges = findEdges(red_highC, 0, 0, 0, fspecial('gaussian', 20, 4), ...
    thresh, 0, 'Canny');

imshow( edges);
%imwrite(imfuse(red_highC,edges), fullfile(strcat(imageHeader, image,'-
edges',imageType)));
```

```
% lowPassSuppression.m
% Driver to adjust image low intensity pixel values and compute basic edge
% detection

close all;
figure;
%Images 20-69
for i = 20:69
iString = num2str(i);
%I = imread(strcat('C:\Users\Turkey\Documents\MATLAB\CIS II Data\rec-000098',
iString, '.bmp'));
I = imread(strcat('/Users/johnlee/Dropbox/Johns Hopkins/Computer Integrated
Surgery II/recimg/rec-000098', iString, '.bmp'));
grayIm = rgb2gray(I);
grayIm = im2double(grayIm);

%Tweak these parameters for preprocessing
%Default filter: Gaussian, 50, 15
filter = fspecial('gaussian', 50, 15);
lowPass = imfilter(grayIm, filter);
highPass = grayIm - lowPass;

%Tweak these parameters for preprocessing
%Default values: 2*high + 0.3*low + 0.3
procIm = 2.0*highPass + 0.3*lowPass + 0.3;

E = imcomplement(imcomplement([grayIm, procIm]));
color = cat(3, zeros(size(E)), ones(size(E)), ones(size(E)));

E = [grayIm, procIm];
% grayIm = imcomplement(grayIm);
% procIm = imcomplement(procIm);
imshow(E, 'initialMag', 100);
hold on;
h = imshow(color);
hold off

%Tweak these parameters for edge detection
thresh = 0.00;
%Default sigma value: 10
sigma = 10;
edgeType = 'canny';
edgesGray = edge(grayIm, edgeType, thresh, sigma);
```

```
edgesProc = edge(procIm, edgeType, thresh, sigma);
set(h, 'AlphaData', [edgesGray, edgesProc]);

pause(0.25)
end
```

```
% PbDriver.m
% Driver to evaluate local edge strength value

close;
imageHeader = 'rec-000098';
imageType = '.bmp';
labeledImageType = '.png';
outputHeader = fullfile('output');
inputHeader = fullfile('input');
image1 = '20';
I1 = imread(fullfile('input', strcat(imageHeader, image1, imageType)));
I2 = imread(fullfile('input', strcat(imageHeader, '22', imageType)));
I3 = imread(fullfile('input', strcat(imageHeader, image1,'-edges',
labeledImageType)));



red = I1(:,:,1);
red2 = I2(:,:,1);

%paramters
red_highC = imadjust(red,[0 0.7],[0 1]);
red_highC2 = imadjust(red2,[0 0.7],[0 1]);
thresh = .000;
filter = fspecial('gaussian', 20, 4);

edges = findEdges(red_highC, 0, 0, 0, filter, ...
    thresh, 0, 'Canny');
false_points = I3(1:end,1:end,1) == 0 & ...
    I3(1:end,1:end,2) >= 254 & ...
    I3(1:end,1:end,3) >= 253;
false_edges = edges & false_points;
true_edges = edges & ~false_points;

%edges2 = findEdges(red_highC2, 0, 0, 0, filter, ...
%    thresh, 0, 'Roberts');


%% display
% E = red_highC;
% color = cat(3, zeros(size(E)), ones(size(E)), ones(size(E)));
% imshow(I1, 'initialMag', 100);
% hold on;
% h = imshow(color);
% hold off
% set(h, 'AlphaData', true_edges);
%
% hold on
```

```
% color = cat(3, ones(size(E)), zeros(size(E)), ones(size(E)));
% h1 = imshow(color);
% set(h1, 'AlphaData', false_edges);

output = mPb(red_highC,edges);
imshow(output);
```

```
% postTrainingAll.m
% Post processing and image morphology operators to distinguish binary edges

close all;
warning('off','all');
tic;
imageHeader = 'rec-000098';
imageType = '.bmp';
for i = 20:69
    I0 = imread(fullfile('input', strcat(imageHeader, num2str(i),
imageType)));
    % labeled0 = imread(fullfile('input', strcat('labeled-', imageHeader,
'57', '.png')));
    disp('Evaluating...');
    [features, edges] = extractFeatures(I0);
    % trueFeatures = extractLearningSet(labeled0);

    predictedLabels = predict(svm, features);
    % Find features labeled with 1 (occluding contour)
    edgeCoords = find(edges == 1);
    featureEdgeCoords = find(predictedLabels == 1);
    featureEdges = edgeCoords(featureEdgeCoords);
    testFeatures = zeros(size(edges));
    testFeatures(featureEdges) = 1;

    % Find features labeled with 0 (not an occluding contour)
    % disp('Finding features labeled with 0');
    % nonFeatureEdgeCoords = find(predictedLabels == 0);
    % nonFeatureEdges = edgeCoords(nonFeatureEdgeCoords);
    % testNonFeatures = zeros(size(edges));
    % testNonFeatures(nonFeatureEdges) = 1;
    se = strel('disk',3);
    cls = imclose(bw,se);
    bw = bwareaopen(testFeatures,15);
    filename = strcat(imageHeader,num2str(i),'-edges',imageType);
    imwrite(bw, filename);
end
% d = imdilate(bw, se);
% for i = 1:1
% %     d = imdilate(d, se);
% end
% morph = bwmorph(d,'thin',inf);
% morph = bwmorph(morph, 'clean');
% trueDist = bwdist(trueFeatures);
% trueDist(~morph) = 0;
% avgDist = sum(sum(trueDist))/size(find(morph==1),1);
```

```
imshow([testFeatures bw]);
[normals,x,y] = getNorms(morph);
% figure;
% imshow(morph);
% hold on;
% xoff = -0.2:.1:0.2;
% for i = 1:length(normals)
%     xplot = x(i) + xoff;
%     yplot = normals(i).*xoff + y(i);
%     plot(xplot,yplot);
% end
% hold off;
title('Test Features vs. NonFeatures');
toc;
```

```
% smoothnessEdgeCombiner.m
% Optical flow driver

close all;

%File name constants
%Make sure to change directory to cis-ii-sinus-surgery sandbox directory
imageHeader = 'rec-000098';
imageType = '.bmp';
outputHeader = fullfile('output');
edgeHeader = fullfile(outputHeader, 'edges');
smoothHeader = fullfile(outputHeader, 'smoothness');
finalHeader = fullfile(outputHeader, 'final');
f = figure;

%Default 20:68
for i = 20:68
%Read in the images
iStr = num2str(i);
iNextStr = num2str(i+1);
I1 = imread(fullfile('input', strcat(imageHeader, iStr, imageType)));
I2 = imread(fullfile('input', strcat(imageHeader, iNextStr, imageType)));

%Preprocess images for HS
filter = fspecial('gaussian', 20, 15);

blur1 = imfilter(I1, filter);
blur2 = imfilter(I2, filter);
% blur1 = medfilt2(rgb2gray(I1), [15 15]);
% blur2 = medfilt2(rgb2gray(I2), [15 15]);

%Perform HS on blurred images
[u, v] = HS(blur1, blur2, 20);

%Calcuate smoothness based on optical flow
s = calcSmoothness(u, v, I1, I2, 20, 95);

%Combine edges from both image
edges = findEdges(blur1) ;%| findEdges(I2);
```

```
%Output on screen and to disk

if mean(mean(s))/max(max(s)) > 0.037
    thresh = 0.09;
elseif mean(mean(s))/max(max(s)) < 0.02
    thresh = 0.0375;
else
    thresh = 0.052;
end
thresh = 0.0375;
output = s ./ max(max(s)) > thresh & edges;

warning('off','all');
fprintf('%d: max: %f, mean: %f, mean/max: %f\n',i,  max(max(s)),
mean(mean(s)), mean(mean(s))/max(max(s)));

imwrite(edges, fullfile(edgeHeader, strcat(imageHeader, iStr,'-
',iNextStr,imageType)));
imwrite(s*5, fullfile(smoothHeader, strcat(imageHeader, iStr,'-
',iNextStr,imageType)));

E = rgb2gray(I1);
color = cat(3, zeros(size(E)), ones(size(E)), ones(size(E)));
imshow(I1, 'initialMag', 100);
hold on;
h = imshow(color);
hold off
set(h, 'AlphaData', output);
saveas(f, fullfile(finalHeader, strcat(imageHeader, iStr,'-
',iNextStr,imageType)));
%imwrite(output, fullfile(finalHeader, strcat(imageHeader, iStr,'-
',iNextStr,imageType)));
pause(0.01);

end
```

```
% svmTrainer.m
% Driver to train the SVM

%File name constants
%Make sure to change directory to cis-ii-sinus-surgery sandbox directory
close all;
imageHeader = 'rec-000098';
imageType = '.bmp';
labeledImageType = '.png';
outputHeader = fullfile('output');
inputHeader = fullfile('input');
N = 10;
xdim = 1280; ydim = 1024; nIchnl = 3; nLchnl = 1;
I = uint8(zeros(ydim,xdim,nIchnl,N));
L = uint8(zeros(ydim,xdim,nLchnl,N));
for i = 1:N
    imnum = 15 + 5*i;
```

```
    imstr = num2str(imnum);
    Itemp = imread(fullfile('input', strcat(imageHeader, imstr, imageType)));
    Ltemp = imread(fullfile('labeled', strcat('labeled-', imageHeader, imstr,
labeledImageType)));
    I(:,:,:,i) = Itemp;
    L(:,:,:,i) = Ltemp;
end
I0 = imread(fullfile('input', strcat(imageHeader, '47', imageType)));

% Train SVM on hand labeled image
disp('Training on hand labeled image');
TSP = logical(zeros(ydim,xdim,N));
for i = 1:N
    TSP(:,:,i) = extractLearningSet(L(:,:,:,i));
end
[svm, trainingFeatures, trainingLabels] = trainSVM(I, TSP);

% Evaluate using a separate image
postTraining;

% Visualize support vectors
% sv = svm.SupportVectors;
% figure
% gscatter(trainingFeatures(:,1), trainingFeatures(:,2),trainingLabels)
% hold on
% plot(sv(:,1), sv(:,2),'ko','MarkerSize',10)
% legend('0','1','Support Vector')
% hold off

% Visualize trainingSetPoints and edges
% E = rgb2gray(I);
% color = cat(3, zeros(size(E)), ones(size(E)), ones(size(E)));
% imshow(I, 'initialMag', 100);
% hold on;
% h = imshow(color);
% hold off
% set(h, 'AlphaData', trainingSetPoints);

% hold on
% color = cat(3, ones(size(E)), zeros(size(E)), ones(size(E)));
% h1 = imshow(color);
% set(h1, 'AlphaData', edges);
% edges = E(svm>=1);
```

SRC

```
% bifurcateCircle.m
% Creates 2 semicircle masks
% INPUT:
% theta = angle in degrees of bifurcation
% rad = radius in pixels of disk
% OUTPUT:
% topMask = binary mask of the top half of the disk
% bottomMask = binary mask of the bottom half of the disk

function [topMask, bottomMask] = bifurcateCircle(theta, rad)
```

```
    theta = deg2rad(theta);
    mask = fspecial('disk',rad)~=0;
    pixelIndex = find(mask);
    [I, J] = ind2sub(size(mask),pixelIndex);

    %determine above and below points
    B = round([rad + 100*cos(theta), rad + 100*sin(theta)]);
    position = sign((B(2)-rad)*(I-rad) - (B(1)-rad)*(J-rad));

    topMask = NaN(size(mask));
    bottomMask = NaN(size(mask));
    topMask(pixelIndex(position == 1)) = 1;
    bottomMask(pixelIndex(position <= 0)) = 1;
    topMask = topMask;
    bottomMask = bottomMask;
end
```

```
% calcSmoothness.m
% Calculates smoothness from optical flow vectors
% INPUT:
% u = x-direction optical flow vector
% v = y-direction optical flow vector
% I1 = first image
% I2 = second image
% w = width
% intThresh = threshold
% OUTPUT:
% s = smoothness map

function s = calcSmoothness(u, v, I1, I2, w, intThresh)
if nargin == 4
    w = 5;
end

if nargin < 6 && nargin >= 4
    intThresh = 90;
end

if size(u) ~= size(v)
    error('u and v must be the same size');
end
if size(I1,3) == 3
    I1 = rgb2gray(I1);
end
if size(I2,3) == 3
    I2 = rgb2gray(I2);
end

if mod(w, 2) == 0
    w = w + 1;
end
%
```

```
% lbHalf = floor(w/2);
% if mod(w, 2) == 0
%     ruHalf = w/2 - 1;
% else
%     ruHalf = floor(w/2);
% end
% s = zeros(size(u));
% su = zeros(w^2, (size(u,1)-2*w)*(size(u,2)-2*w));
% sv = zeros(w^2, (size(u,1)-2*w)*(size(u,2)-2*w));
% count = 1;
% for i = 1+lbHalf:size(u,1)-ruHalf
%     for j = 1+lbHalf:size(u,2)-ruHalf
%         if I1(i,j) > intThresh || I2(i,j) > intThresh
%             su(1:end, count) = zeros(w^2,1);
%             sv(1:end, count) = zeros(w^2,1);
%         else
%             su(1:end, count) = reshape(u(i-lbHalf:i+ruHalf,j-
lbHalf:j+ruHalf), [w^2,1]);
%             sv(1:end, count) = reshape(v(i-lbHalf:i+ruHalf,j-
lbHalf:j+ruHalf), [w^2,1]);
%         end
%         count = count + 1;
%     end
% end




% sAngle = atan(sv./su);
% sAngle = reshape(var(sAngle./max(max(sAngle))), [j-lbHalf, i-lbHalf])';
% sMag = sqrt(su.^2+sv.^2);
% sMag = reshape(var(sMag./max(max(sMag))), [j-lbHalf, i-lbHalf])';
% s2(1+lbHalf:size(u,1)-ruHalf, 1+lbHalf:size(u,2)-ruHalf) = (sAngle+sMag)/2;
% su = reshape(var(su), [j-lbHalf, i-lbHalf])';
% sv = reshape(var(sv), [j-lbHalf, i-lbHalf])';
% s(1+lbHalf:size(u,1)-ruHalf, 1+lbHalf:size(u,2)-ruHalf) = (su+sv)/2;

su = stdfilt(u, ones(w));
sv = stdfilt(v, ones(w));
mask = I1 < intThresh & I2 < intThresh;

s = (su + sv)/2.*mask;

end
```

```
% cropImg.m
% Crops the image using a mask
% INPUT:
% img = image
% x = x coordinate of center of mask
% y = y coordinate of center of mask
% rad: radius in pixels of mask
% mask = binary mask to determine which places to cut
% OUTPUT:
```

```
% crop = cropped image

function crop = cropImg(img, x, y, rad, mask)
    sz = size(img);
    if nargin < 2
        x = floor(sz(2)/2);
        y = floor(sz(1)/2);
        rad = 8;
        mask = uint8(fspecial('disk',rad)~=0);
    elseif nargin == 4
        mask = uint8(fspecial('disk',rad)~=0);
    end
    x0 = x - rad;
    x1 = x + rad;
    y0 = y - rad;
    y1 = y + rad;


    if isa(mask,'uint8')
        mat = uint8(zeros(2*rad + 1));
    else
        mat = zeros(2*rad + 1);
    end

    if x0 < 1 && y0 < 1
        xoff = -x0 + 1;
        yoff = -y0 + 1;
        mat(yoff+1:end,xoff+1:end) = img(1:y1,1:x1);
    elseif x0 < 1 && y1 > sz(1)
        xoff = -x0 + 1;
        yoff = -(y1 - sz(1));
        mat(1:end+yoff,xoff+1:end) = img(y0:end,1:x1);
    elseif y0 < 1 && x1 > sz(2)
        xoff = -(x1 - sz(2));
        yoff = -y0 + 1;
        mat(yoff+1:end,1:end+xoff) = img(1:y1,x0:end);
    elseif x1 > sz(2) && y1 > sz(1)
        xoff = -(x1 - sz(2));
        yoff = -(y1 - sz(1));
        mat(1:end+yoff,1:end+xoff) = img(y0:end,x0:end);
    elseif x0 < 1
        xoff = -x0 + 1;
        mat(1:end,xoff + 1:end) = img(y0:y1,1:x1);
    elseif y0 < 1
        yoff = -y0 + 1;
        mat(yoff + 1:end,1:end) = img(1:y1,x0:x1);
    elseif x1 > sz(2)
        xoff = -(x1 - sz(2));
        mat(1:end,1:end + xoff) = img(y0:y1,x0:end);
    elseif y1 > sz(1)
        yoff = -(y1 - sz(1));
        mat(1:end + yoff,1:end) = img(y0:end,x0:x1);
    else
        mat = img(y0:y1,x0:x1);
    end
    crop = mat.*mask;
```

```
end
```

```
% extractFeatures.m
% Extracts a set of features from an image at the given points
% INPUT:
% I = the given image
% points = a binary array that signifies which points in I to extract
% features from
% OUTPUT:
% features = feature vector for SVM
% points = a binary array that signifies where the feature vectors come
% from
function [features, points] = extractFeatures(I, points)
if nargin == 1
    % If no points given, use results of our edge detection algorithm as
points
    filter = fspecial('gaussian', 20, 4);
    thresh = 0;
    red = I(:,:,1);
    red_highC = imadjust(red,[0 0.7],[0 1]);
    points = findEdges(red_highC, 0, 0, 0, filter, ...
    thresh, 0, 'Canny');
end
if size(I,3) == 3
   grayIm = im2double(rgb2gray(I));
end

scaledGradient = extractScaledGrad(grayIm, points);
hist = extractFeaturesHist(I, points);
mPb = extractMPb(grayIm, points);
features = [scaledGradient hist mPb];
end
```

```
% extractFeaturesHist.m
% Extracts histogram feature from pixel using a circular mask
% INPUT:
% I = the given RGB image
% points = a binary array that signifies which points in I to extract
% features from
% OUTPUT:
% features = matrix of histogram values
% points = edge points that were calculated

function [features, points] = extractFeaturesHist(I, points)

if nargin == 1
    % If no points given, use results of our edge detection algorithm as
points
    points = findEdges(I);
end
r = 16;
bnum = 16;
% Current features: hue, saturation, value (intensity)
Ib = rgb2gray(I);
```

```
edgepts = find(points > 0);
hists = zeros(floor(length(edgepts)), bnum);
count = 1;
mask = uint8(fspecial('disk',r)~=0);
for i = 1:length(edgepts)
    [y,x] = ind2sub(size(Ib),edgepts(i));
    window = cropImg(Ib,x,y,r,mask);
    w = double(window(:));
    h = hist(w,bnum);
    h = h/max(h);
    hists(count,:) = h;
    count = count + 1;
end
features = hists;
end
```

```
% extractLearningSet.m
% Extracts non zero points from labeled image
% INPUT:
% labeledImage = mono-color image
% OUTPUT:
% poitns = extracted nonzero points

function points = extractLearningSet(labeledImage)
I = labeledImage;
points = I > 0;
end
```

```
% extractMPb.m
% Extracts the local edge strength for SVM
% INPUT:
% I = the given RGB image
% points = a binary array that signifies which points in I to extract
% features from
% OUTPUT:
% features = feature vector
% points = point vector

function [features, points] = extractMPb(I, points)

if nargin == 1
    % If no points given, use results of our edge detection algorithm as
points
    points = findEdges(I);
end
if size(I,3) == 3
   I = rgb2gray(I);
end

multiPb = mPb(I, points);
%load('multiPb img20.mat');
%multiPb = multiPb3;
multiPb = multiPb(points);
```

```
features = multiPb;
end
```

```
% extractScaledGrad.m
% Extracts SVM features for the scaled gradient
% INPUT:
% I = the given RGB image
% points = a binary array that signifies which points in I to extract
% features from
% OUTPUT:
% features = feature vector
% points = point vector

function [features, points] = extractFeaturesScaledGrad(I, points, scale)
if nargin == 2
   scale = 0.2;
end
gradient = imgradient(imresize(I, scale));
gradient = imresize(gradient, 1/scale, 'lanczos3');
features = gradient(points);
end
```

```
% filterSmoothness.m
% Filter input to eliminate values below the threshold
% INPUT:
% s = smoothness map
% thresh = threshold
% OUTPUT:
% fs = filtered smoothness map

function fs = filterSmoothness(s, thresh)
    fs = s;
    for i = 1:size(fs,1)
        for j = 1:size(fs,2)
            if fs(i,j) < thresh
                fs(i,j) = 0;
            end
        end
    end
end
```

```
% findEdges.m
% Performs preprocessing and built in MATLAB edge detection
% INPUT:
% I = image
% lowPassFactor = low intensity pixel threshold
% highPassFactor = high intensity pixel threshold
% boost = overall image intensity boost
% filter = preprocessing filter for image
% thresh = edge detection threshold
% sigma = edge detection sigma
% edgeType = name of edge detection to be used
```

```
% OUTPUT:
% edges = binary edge map

function edges = findEdges(I, lowPassFactor, highPassFactor, boost, ...
    filter, thresh, sigma, edgeType)

if nargin == 1
   lowPassFactor = 0.3;
   highPassFactor = 2;
   boost = 0.3;
   filter = fspecial('gaussian', 20, 4);
   thresh = 0;
   edgeType = 'C';
end
I = imfilter(I, filter);
if size(I,3) == 3
    grayIm = rgb2gray(I);
else
    grayIm = I;
end
grayIm = im2double(grayIm);

%Tweak these parameters for preprocessing
%Default filter: Gaussian, 50, 15
%lowPass = imfilter(grayIm, filter);
%highPass = grayIm - lowPass;

%Tweak these parameters for preprocessing
%Default values: 2*high + 0.3*low + 0.3
%procIm = highPassFactor*highPass + lowPassFactor*lowPass + boost;
procIm = medfilt2(grayIm, [15 15]);

%Tweak these parameters for edge detection
%Default threshold: 0
%Default sigma value: 10

edges = edge(procIm, edgeType);
end
```

```
% getNorms.m
% Computes normal vectors for a given point on an edge
% INPUT:
% points = binary array of points
% OUTPUT:
% norms = normal vectors
% xpos = xposition of vector origin
% ypos = yposition of vector origin

function [norms, xpos, ypos] = getNorms(points)
points = uint8(points);
    r = 8;
    edgepts = find(points > 0);
    mask = uint8(fspecial('disk',r)~=0);
    norms = zeros(length(edgepts),1);
```

```
    xpos = zeros(length(edgepts),1);
    ypos = zeros(length(edgepts),1);
    for i = 1:length(edgepts)
        [y,x] = ind2sub(size(points),edgepts(i));
        window = cropImg(points,x,y,r,mask);
        w = double(window(:));
        ep = find(w > 0);
        [epy,epx] = ind2sub(size(mask),ep);
        p = polyfit(epx,epy,1);
        slope = p(1);
        norms(i) = -1/slope;
        xpos(i) = x;
        ypos(i) = y;
    end
end
```

```
% mPb.m
% Calulates local edge strength for an image at all edge points using a
% range of bifurcation angles
% INPUT:
% image = mono-color image to be processed
% points = matrix of edge points to evaluate at
% OUTPUT:
% PbArray = local edge strength map

function [PbArray] = mPb(image, points)
tic;

%Images: images to be processed
%points: binary array
image = im2double(image);
imageSize = size(image);
[centerIndex(:,2), centerIndex(:,1)] = find(points);

%narrow sampled points
numToSkip = 0;
numOfAngles = 8;
rad = 20;
%

centerIndex = centerIndex(1:numToSkip+1:size(centerIndex,1), :);
index = sub2ind(imageSize, centerIndex(:,2), centerIndex(:,1));


%iterate through each theta and find greatest chi square
%distance
theta = 0:(180/numOfAngles):180;
PbArray = zeros(size(image));
for k = 1:numOfAngles
    currTheta = theta(k);
    [topMask, botMask] = bifurcateCircle(currTheta,rad);
    for j = 1:size(centerIndex,1);
        currCenter = centerIndex(j,:);
        currIndex = index(j);
```

```
        var = Pb(topMask, botMask ,currCenter,image);
        if var > PbArray(currIndex)
            PbArray(currIndex) = var;
        end
    end
end
toc;
end
```

```
% Pb.m
% Calculates local edge strength
% INPUT:
% topMask = binary mask for one histogram (at a given angle)
% botMask = binary mask for second histogram (at a given angle)
% center = center coordinates
% image = image
% OUTPUT:
% variance = edge strength value

%If pixel coordinates are out of bounds, returns variance of 0
%center: [x y] coordinates of point to investigate
%theta: bifurcating angle in degrees
function [variance] = Pb(topMask, botMask, center, image)

%Variables
radius = 20;
binNumber = 25;
%

binSize = (0: 1/(binNumber-1):1);
imageSize = size(image);
x = center(1);
y = center(2);
variance = 0;

%skips edge cases
if any([center<= radius, center >= (imageSize - radius)])
    return;
end

rad = floor(size(topMask,1)/2);
tValue = cropImg(image, x, y, rad, topMask);
bValue = cropImg(image, x, y, rad, botMask);

%compute gradient
tValue(tValue == 0) = NaN;
bValue(bValue == 0) = NaN;

[tCounts, ~] = hist(tValue, binSize);
tCounts = tCounts/sum(tCounts);
[bCounts, ~] = hist(bValue, binSize);
bCounts = bCounts/sum(bCounts);
for i = 1:binNumber
    %only evalutes nonzero denominators
```

```
    if tCounts(i) + bCounts(i) == 0
    else
        variance = variance + (((tCounts(i) - bCounts(i))^2) / (tCounts(i) +
bCounts(i)))*.5;
    end
end


% display
% close all;
% bar([tCounts, bCounts]);
% figure;
% imshow(imfuse(cropTop, cropBot));
end
```

```
function [svm, trainingFeatures, trainingLabels] = runSVM(I, edges, svm)
%piece of svmTrainer

% Evaluate using a separate image
[features, edges] = extractFeatures2(I, edges);
predictedLabels = predict(svm, features);

figure;
% Find features labeled with 1 (occluding contour)
edgeCoords = find(edges == 1);
featureEdgeCoords = find(predictedLabels == 1);
featureEdges = edgeCoords(featureEdgeCoords);
testFeatures = zeros(size(edges));
testFeatures(featureEdges) = 1;

% Find features labeled with 0 (not an occluding contour)
nonFeatureEdgeCoords = find(predictedLabels == 0);
nonFeatureEdges = edgeCoords(nonFeatureEdgeCoords);
testNonFeatures = zeros(size(edges));
testNonFeatures(nonFeatureEdges) = 1;

figure;
% Visualize trainingSetPoints and edges
E = I;
color = cat(3, zeros(size(E)), ones(size(E)), ones(size(E)));
imshow(I, 'initialMag', 100);
hold on;
h = imshow(color);
hold off
set(h, 'AlphaData', testFeatures);

hold on
color = cat(3, ones(size(E)), zeros(size(E)), ones(size(E)));
h1 = imshow(color);
set(h1, 'AlphaData', testNonFeatures);
```

```
% trainSVM.m
```

```
% Trains an SVM based on an edges in an image and a training set of points
% INPUT:
% I = the original image
% TSP = a binary array of ground truth points
% OUTPUT:
% svm = trained support vector machine
% trainingFeatures = collection of feature values
% trainingLabels = positive and negative labels for trainingFeatures

function [svm, trainingFeatures, trainingLabels] = trainSVM(I, TSP)
trainingFeatures = [];
nonFeatures = [];
sz = size(I);
len = sz(4);
filter = fspecial('gaussian', 20, 4);
thresh = 0;
for i = 1:len
    TFstr = 'Extracting training feature ';
    TFstr = strcat(TFstr, num2str(i),'.');
    disp(TFstr);
    TF = extractFeatures(I(:,:,:,i), TSP(:,:,i));
    trainingFeatures = [trainingFeatures; TF];
end
trainingLabels = ones(size(trainingFeatures, 1), 1);
for i = 1:len
    TFstr = 'Extracting non-feature ';
    TFstr = strcat(TFstr, num2str(i),'.');
    disp(TFstr);
    red = I(:,:,1,i);
    red_highC = imadjust(red,[0 0.7],[0 1]);
    nFE = findEdges(red_highC, 0, 0, 0, filter, thresh, 0, 'Canny') &
~TSP(:,:,i);
    NF = extractFeatures(I(:,:,:,1), nFE);
    nonFeatures = [nonFeatures; NF];
end
trainingFeatures = [trainingFeatures; nonFeatures];
trainingLabels = [trainingLabels ; zeros(size(nonFeatures,1), 1)];
disp('Training SVM');
tic;
svm = fitcsvm(trainingFeatures(1:2:end,:), trainingLabels(1:2:end), ...
    'KernelFunction', 'rbf');
toc;
end
```