Final Report

Lung Fissure Segmentation in CT Images

ECE 5780

Group 3

Ryan Ashley
Clifford Chou
Caleb Woo

Abstract

Fissures are macroscopic structures that separate the the lobes of the left and right lungs. The segmentation of lung fissures is critical in the analysis of CT lung images as it allows for the localization of pathologic tissue such as nodules or embolisms in the lobes of the lung. A variety of techniques for segmenting lung fissures are shown in the literature including curve fitting to pre-existing lung models, finding global minimal paths and applying derivative of stick filters. Segmentation of fissures is challenging primarily due to their low contrast appearance in CT images and presence of many surrounding structures that can mistaken for fissures. Recently developed methods using watershed transformation and small segment removal show promise for accurately segmenting lung fissures. A sample of fissures from multiple lung CT images will be marked for ground truth. The segmentation algorithm will then be applied to these images and the accuracy of the segmentation can be determined by comparison to the ground truth markings.

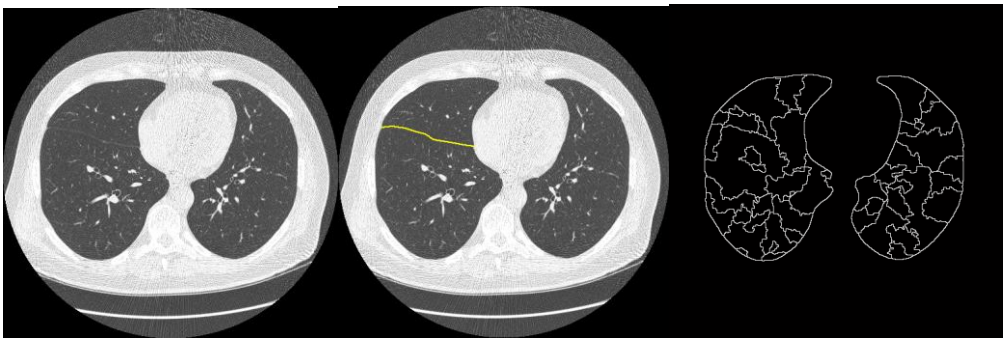

Fig 1.1 Original CT scan

Fig 1.2 CT scan with marked ground truth

Fig 1.3 Segmented Image

Introduction

The human lungs have three major fissures dividing both the left and right lungs into lobes. The left lung has an oblique fissure that separates the inferior and superior lobes of the lung while the right lung has two fissures that divide the organ into inferior, middle and superior lobes known as the oblique and horizontal fissures respectively. These fissures are the structures that physically separate the lobes and are the interfaces of the pleural surfaces of each lobe.

Segmentation of lung fissures allows surgeons to easily determine the locations of pathologic tissue relative to the overall structure of the lungs from the CT image data. This information is invaluable in the preparation and process of surgery by facilitating the removal or treatment of diseased tissue. The primary challenges in the segmentation of lung fissures is their low contrast appearance in CT images due to their structure being defined as absence of tissue. This issue is often compounded by the significant noise present in most low dose CT images that blends the fissure surface into the background of the image [1]. Additionally, the many other anatomical structures such as blood vessels, airways and nodules can be mistaken for fissures and make fissure segmentation more difficult.

Previous work to segment lung fissures have used techniques such as derivative of stick filters to find the deviation of intensities along a line to detect fissures and suppress segmentation of large structures [1]. Other research has used Bayesian networks to grow a curve that would fit the fissure using prior knowledge of fissure geometry and a ridge map of the image [2] Additionally, there has been work done using energy functions to find global minimal paths across the lungs that correspond to the lung fissure [3]. This project will be most closely related to recent work done that first performs a watershed transformation on a gradient projection of the image to segment the fissures among other structures, and then using a small segment removal algorithm to isolate the fissures [4].

In order to segment the lung fissures a gradient operator can first be applied to the image to obtain a gradient magnitude. Internal and external markers can then be found for use in a following marker based watershed algorithm. The internal object markers are found by performing an opening and a closing on the original image and then thresholding to find maxima that are defined as objects. The external background markers can then be found by thresholding the original image and finding the connected background components that serve as seed points for the watershed. The watershed transformation can then be applied to find objects marked by high intensity gradients and the accessory fissures and other structures can be removed by setting a threshold number of pixels that must be contained in an object [1].

This algorithm was developed for use with single 2D axial slices of conventional CT images in the original paper by Davaki et al. [1]. This project was originally intended to implement a 3D variant of this algorithm, although do to the time constraints and significant complexity, an algorithm based on the original 3D algorithm was implemented.

The results of the test and algorithm were not positive. The hypothesis and set goal was to achieve an RMSD value of 1.0 mm relative to marked ground truth. However, the resulting segmentation from the algorithm was not capable of segmenting the fissures to a point where it was impossible to even determine an RMSD value. There were some drawbacks to the algorithm that were not fully considered which caused segmentation issues. There are several improvements that could be implemented that can potentially fix the encountered problems.

Materials and Methods

The initial step is to perform a gradient operator on the original CT image. A few adjustments were made here though. Using lung masks which were part of the provided data set, the CT scan was narrowed down to solely the lungs to avoid processing structures outside of the lungs themselves. The focus of the fissure is within the lung so the outside structures only add noise to the objects within the lung. Since the black background will also affect the various thresholding done to determine markers, which will be discussed below, the background was not included in calculating thresholds. The gradient operator was done via image pre-processing using the VisionX command *vsobel*.

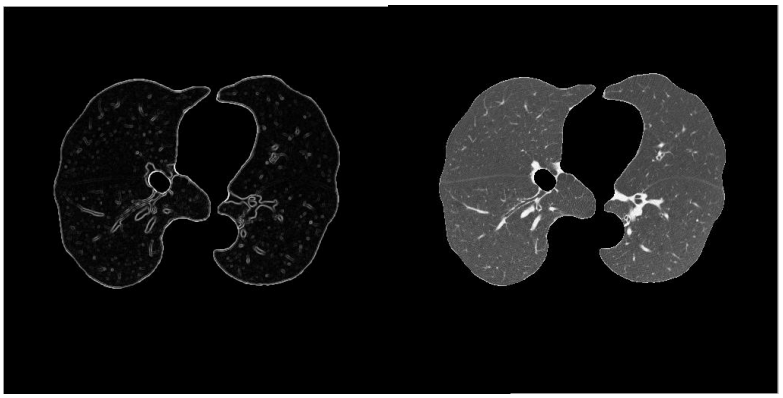

Fig 2.1 Gradient of adjusted CT Fig 2.2 CT manipulated with lung masks

The main purpose of this particular method is to enhance the fissure intensity to help distinguish it from the much more visible airways and vessels as well as to prevent oversegmentation with the watershed transform. The markers were key in this aspect. Internal markers use the opening and closing to widen the peaks and valleys (high and low intensities) and then use 8-point connected components to determine regional maxima. The connected components allows for a certain range of pixel intensity value to help account for the varying intensity of the lung fissure as well as other features in the lung. The same concept is applied to the external markers but to create seed points or catchment basins for the watershed. This is done through applying a threshold on the original image and then using a distance transform to thin out the objects. Finally use a watershed transform was used to narrow down the objects to a minimum. The opening and closing morphological operations were performed with VisionX function *vmorph* and the resulting image was used as an input in order to calculate the regional minima and threshold. A parameter that was trained to optimize the internal marker results was the kernel size for the opening and closing, *s1* and *s2* respectively.

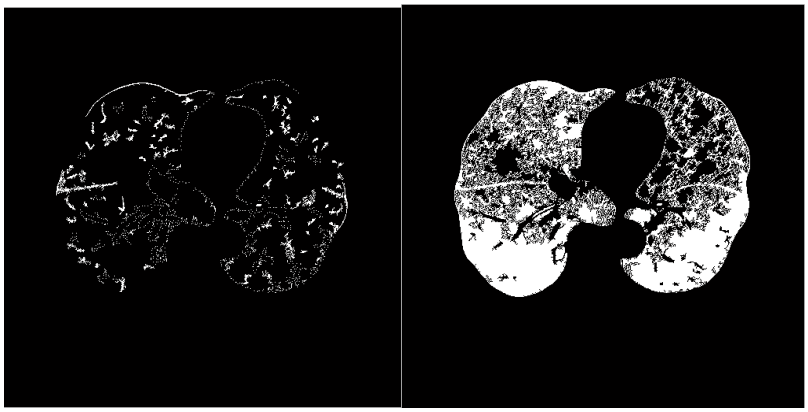

Fig 2.3 Internal Markers Fig 2.4 Thresholded image, for external markers

The watershed was performed on the gradient image with superimposed internal and external marker images. The regional maxima from the internal markers illustrate the location of the objects within the lung which prevents oversegmentation because the objects are now marked and cannot be divided. Using this with the gradient image helps identify sharp changes in

intensity which, in conjunction with the internal markers, can identify only the objects. The external markers indicate the minima where the watershed is seeded from.

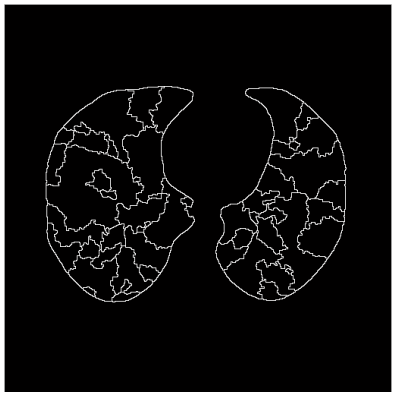

Fig 2.5 Final Segmentation result with lung outline

The resulting image from the watershed on these superimposed images will have all objects segmented including the fissure. An assumption is made here that the fissure is the longest and largest connected component that is segmented when the lung boundary is removed by a mask. Through the training set, it is possible to determine the optimal threshold *t* where all the small segments such as vessels and airways are eliminated but the fissure it preserved. It would have also been possible to simply search for the largest connected component and then eliminate the rest of the components but that method is slower as it requires two iterations through the entire image, one to determine the longest connected component and the other to eliminate the other components.

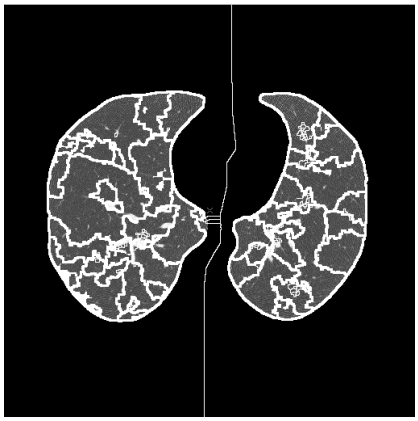

Fig 2.6 Segmentation superimposed on original image

A series of manually segmented fissures from multiple lung low dose CT images will be used to test the algorithm. The success of this approach can be quantified by comparing the number of matching pixels between the algorithm and the ground truth segmentation and error can be found by calculating the distance of nonmatching pixels from the manually segmented fissure. Based on the findings of the Devaki et al. paper, a root mean squared error on the order of 1 mm is achievable [1]. The root mean squared error will be calculated with the indicated expression.

$$\text{RMSD} = \sqrt{\frac{1}{n}\sum_{i=0}^{n}((x_2-x_1)^2 + (y_2-y_1)^2)} \quad \text{Eq. 1}$$

In order to calculate this for many images, a C program was written to calculate the RMSD value for each tested image. It would search the test result image for the segmentation and save the pixel coordinate of each segmented pixel. It would start from the upper left of the image and go down the columns and then do the same for each row thereafter. The same thing is done for the ground truth image but instead of looking for high pixel values, it looks for the specific color yellow which was used to mark the ground truth. Since the progression of iterating through the image was done the same, it can be assumed that each coordinate saved in the array can correspond to the same coordinate in the opposing image. Using the x and y coordinates, an RMSD value can be calculated. If a segment is longer than the other, the excess coordinates are just compared against the last coordinate in the other image. The RMSD result is in units of pixels though and since every image used has a varying magnification, the conversion to millimeters had to be done manually by checking the size of the CT scan and sizing it relative to the image size (512x512) pixels.

The dataset used to train and test the images are from the Cornell Vision and Image Analysis Group project database. The training set consisted of 10 sets of low dose lung CT scan images (W0001, W0002, W0003, W0004, W0005, W0006, W0007, W0009, W0010, W0021). The test set is also composed of 10 sets of low dose lung CT scans (W0022, W0023, W0024, W0025, W0026, W0027, W0028, W0029, W0030, W0031). Lung masks for each of these sets of images were also used and vital to the project.

*Note - The images above are from a variety of test results

Results

The results of the algorithm were not consistent with the hypothesis. The goal to achieve a RMSD value of 1.0 mm was not met and could not even be tested for because of the poor quality of the segmentation. The fissure segmentation itself did not have results that far off the hypothesis. However, the over segmentation of the vessels and airways as well as the program connecting all the segmentations into a single connected component did not allow for exhaustive and accurate testing. Training was still done to optimize the parameters for opening and closing kernel size. The threshold for connected component size was not needed because of the inability for the algorithm to actually separate the small segments unfortunately. Through the ten training images, the optimal kernel size was determined to be a square kernel of 3x3 pixel size. This was done through observing the final segmentation, whether it was over or under segmented. Because the results could not be quantified in an RMSD value, it was difficult to quantitatively support the optimal kernel size. However, from the following images, it is clear that a kernel size of 3x3 would be best because of the actual fissure segmentation and the amount of other objects that were segmented as well. Intermediary steps to determine optimal kernel size were not conclusive and could not definitely define which one would give the best result.

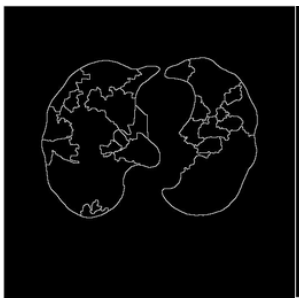

Fig 3.1 2x2 kernel

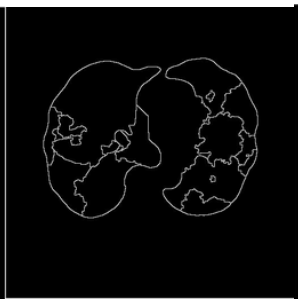

Fig 3.2 3

Fig 3.3 5x5 kernel

From the above images of the final segmentation, it is clear that any kernel above 5x5 would result in under segmentation and any kernel smaller than 2x2 would be significant over segmentation. The fissure segmentation in Fig 3.1 and 3.2 are very similar so the only difference is in the vessel and airway segmentation.

The failure of the algorithm can be attributed to many different factors that are discussed in the next section but there were a few key types of failure that should be noted:

1. The segmentation did not fit the fissure well. The fissure was segmented in the proper location but it did not match the shape due to the algorithm detecting the noise rather than the fissure. This failure was not very common but still resulted in some images.

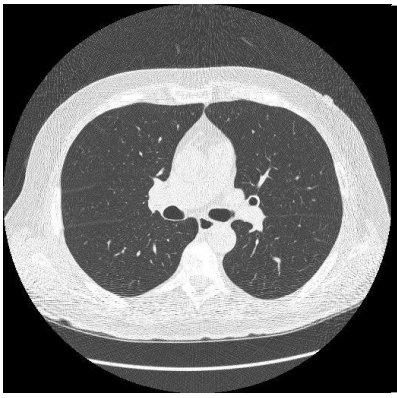

Fig 3.4 Original CT image

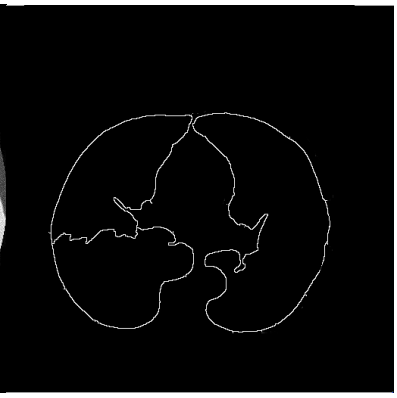

Fig 3.5 Failed segmentation

2. Segmentation did not find the fissures. The fissure was simply not found by the segmentation, instead other objects were found.

Fig 3.6 Failed Segmentation: no fissure

3. Many objects were segmented. Too many objects were segmented and all the objects were connected. This occurred in the majority of cases and the RMSD criteria could not be applied to these images.

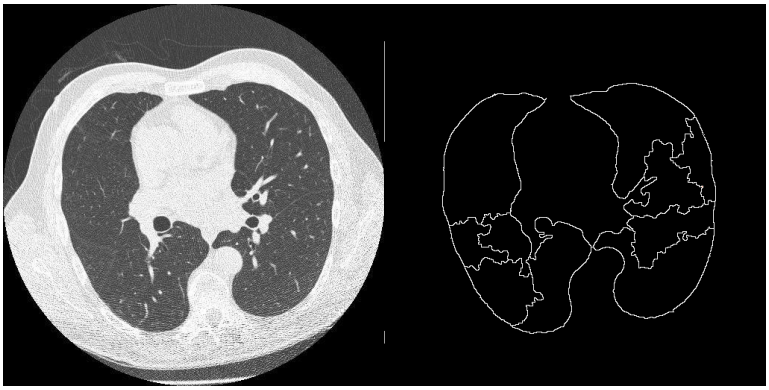

Fig 3.7 Failed Segmentation: too many objects segmented

These three different types of segmentations made it impossible to test the evaluation criteria with the C program so no concrete RMSD values were calculated to compare with the hypothesis or previous work. However, it is possible to manually calculate an RMSD value while manually eliminating the over segmented vessels and airways. It is basically impossible as a human to calculate each pixel so using a rough estimation by calculating the RMSD between segments of the ground truth versus the segmented image, a result of 1.3mm RMSD was achieved. This result is ignoring the over segmented parts as well as ignoring the missing parts of the fissure so it is very inaccurate and generalized.

Discussion

The failure of the algorithm to correctly segment the lung fissures in most cases can be accounted for by the effects of each of the original issues discussed in segmenting lung fissures from low dose CT images. The intensity of significant noise present in the images was often similar to the intensity of the fissures in both the original and gradient images. This noise could be effectively removed by using a standard noise removal filter as it significantly decreases the edge of the fissures. The original paper that used the watershed algorithm for segmenting fissures, did so using conventional CT images, which have significantly less noise than low dose CT images. With this different general data set, fissure segmentation would have been significantly easier as noise is less likely to cause vessels, airways and salt and pepper noise spikes to become connected components that could be equal to or longer than the length of a fissure. The fissure's identity as the longest connected component within the lung was critical to the outcome of the algorithm. The effect of other structures being labeled as connected components was amplified by the algorithm's detection of fissures often not as single components, but as many separated objects due to the fissure's variable intensity over its length.

Despite these issues, the algorithm should have been more effectively if it was implemented in 3D, using the entire CT scan rather than just a single 2D axial slice. Fissures are smooth objects that usually cut the through the entire volume of the lung as a border between the lobes. The detection of fissures is inherently a 3D problem and reducing it to single 2D slices loses a significant amount of very useful information. Normally, whole lung fissures are much larger than any other object within the lungs and in the 3D case, it would be unlikely that other objects would be labeled as being larger, even if they are improperly connected together. Interestingly, many examples of previous work treat fissures as a 2D problem, using single 2D axial slices to find the fissures [1,2,9]. While some recent work does treat the segmentation of fissures as a 3D problem, this is not the overwhelming majority of the research being conducted, suggesting the youth of this problem in computer aided diagnosis [8].

Due to a significant amount of work being done in solving the lung fissure problem in 2D, the dominant evaluation criteria in the literature is the 2D RMSD criteria. This criteria is problematic for several reasons, the first of which being that fissures in the plane of the axial slice do not appear as lines, but rather large open regions. The RMSD criteria requires a line for evaluation and in this situation, no clear ground truth can be defined. Additionally, this criteria does not have any way to deal with segmentations that have more or less marked fissures than the ground truth, which prevented the calculation of this criteria for all of the images in our data set. Finally, this criteria is a poor measure for 3D algorithms as it does not consider the 3D geometry is any way. Other metrics have arisen in the use of 3D algorithms but a set standard evaluation criteria has yet to be defined.

There are many techniques that could be used to expand on the algorithm. The algorithm itself has a lot of potential but two key areas that were problematic were determining the background markers and finding the regional maxima. Adjusting the thresholding was helpful in the

segmentation but the fissure is still too similar to a lot of the background and was not as intense as the noise as well as vessels and airways. Applying a max/min or a mean/median filter may be able to tone down some of the noise and help smooth out the background. The distance transform used to calculate the background markers were difficult to achieve because the thresholding gave a large portion of objects. Creating a skeleton of the objects rather than the background created wild background markers that did not correspond to the original image. This can be addressed by creating an adjusted distance transform and trying different distance transforms, such as chess board and city block rather than just Euclidean, and then applying an adjusted watershed on the distance transformed image to create the skeleton. These are things to explore in future iterations of the algorithm. Depending on the internal and external marker images, the watershed could be more potent in segmenting out the fissures. Another concept would be narrow the fissure window. Instead of checking for the entire lung, use the previous work to segment out the very intense vessels and airways. With the knowledge that fissures only exist where the vessels and airways do not, the window to find the fissures can be narrowed down to the area where vessels and airways were not segmented. The algorithm could then be applied to this space instead of the entire lung.

Conclusion

The 2D marker based watershed algorithm implemented was insufficient to segment the fissures in low dose CT images. The RMSD evaluation criteria could not be calculated in many images due to oversegmentation. In the images that were possible to evaluate, an average RMSD of 1.3 mm was calculated that caused the rejection of the hypothesis that an RMSD of 1.0 mm or better could be achieved based on previous work. The implemented algorithm had numerous issues when used with our data set, but several improvements could be made to achieve satisfactory performance. The algorithm could be implemented in 3D as originally intended due to the 3D nature of the lung fissure problem. Additionally, the creation of internal and external markers could be improved as they are an essential component of the algorithm. Overall, this approach did not successfully solve the fissure problem, and other more robust methods may be more suitable for clinical applications.

References

[1] Devaki, K., & Bhaskaran, V. M. (2014). A NOVEL APPROACH TO DETECT FISSURES IN LUNG CT IMAGES USING MARKER-BASED WATERSHED TRANSFORMATION. Journal of Computer Science , 10(6), 896-905.

[2] Wang, J., Betke, M., & Ko, J. P. (2006). Pulmonary fissure segmentation on CT. Medical Image Analysis, 10(4), 530-547.

[3] Appia, V., Patil, U., & Das, B. (2011). Lung Fissure detection in CT Images using Global Minimal Paths. Proc. of SPIE, 7623.

[4] Xiao, C., Staring, M., & Wang, J. (2011). A Derivative of Stick Filter for Pulmonary Fissure Detection in Lung CT Images.

[5] Kommuru, H. (2013). Pulmonary fissures and lobar variations in relation to surgical & radiological implications. IOSR Journal of Dental and Medical Sciences, 5(1), 51-54.

[6] N.kumar, S., and M. Marsaline Beno. "Segmentation of Lung Lobes and Fissures for Surgical Pre Planning." International Journal of Computer Applications 51.9 (2012): 12-16.

[7] Schmidt-Richberg, Alexander, Jan Ehrhardt, and Matthias Wilms. "Pulmonary Lobe Segmentation with Level Sets." Proc. of SPIE 8314.83142V (2012): 1-8.

[8] Ukil, Soumik, and Joseph M. Reinhardt. "Smoothing Lung Segmentation Surfaces in Three-dimensional X-ray CT Images Using Anatomic Guidance1." *Academic Radiology* 12.12 (2005): 1502-1511.

[9] Qiao, W., Y. Hu, J.H. MacGregor and G. Gelfand, 2008. Segmentation of lung lobes in clinical CT images. Int. J. CARS. 3: 151-163.

Computer Programs

```
/*************************************************************************/
/* vtpeak:     Threshold image between two most sig. hgram peaks     */
/*************************************************************************/

#include "VisXV4.h"          /* VisionX structure include file       */
#include "Vutil.h"           /* VisionX utility header files         */
VisXfile_t *VXin,            /* input file structure                 */
         *VXout;             /* output file structure                */
VisXelem_t *VXlist,*VXptr;   /* VisionX data structure               */
VXparam_t par[] =            /* command line structure               */
{
{  "if=",   0,   " input file, vtpeak: threshold between hgram peaks"},
{  "of=",   0,   " output file "},
{  "-v",    0,   "(verbose) print threshold information"},
{   0,      0,   0} /* list termination */
};
#define  IVAL   par[0].val
#define  OVAL   par[1].val
#define  VFLAG  par[2].val

main(argc, argv)
int argc;
char *argv[];
{

VisXimage_t im;                  /* input image structure          */
int        i,j;                  /* index counters                 */

   int hist[256];                /* histogram bins                 */
   int thresh;                   /* threshold                      */
   int dist;                     /* minimum distance between maxima   */
   int avg1,avg2;                /* averages of Region1 and Region2   */
   int old1,old2;        /* saved averages from previous iteration */
   int total1=0,total2=0;        /* for computing average     */
   int number1=0,number2=0;


   VXparse(&argc, &argv, par);    /* parse the command line         */
   VXin  = VXopen(IVAL, 0);       /* open input file                */
   VXout = VXopen(OVAL, 1);       /* open the output file           */

/************ End of Parameter and initialization section ************/
```

```
   while((VXlist = VXptr = VXreadframe(VXin)) != VXNIL){ /* every frame */
      VXfupdate(VXout, VXin); /* update global constants */
          /* find next byte image */
      while (VXNIL != (VXptr = VXfind(VXptr, VX_PBYTE)))  {
         VXsetimage(&im, VXptr, VXin); /* initialize input structure */

/****************** Application specific section **********************/

         /* clear the histogram */
         for (i = 0; i < 256; i++) hist[i] = 0;

         /* compute the histogram */
         for (i = im.ylo; i <= im.yhi; i++){
            for (j = im.xlo; j <= im.xhi; j++){
               hist[im.u[i][j]]++;
                  }
            }

         /* compute the threshold */
             for (i=0; i <256; i++) {
                    number1=number1+hist[i];
                    total1=total1+i*hist[i];
         }
             thresh=total1/number1;
             fprintf(stderr, "thresh = %d\n",thresh);
             total1=0;
             number1=0;
             for (j=0; j<=10000; j++){
                      for (i=0; i<thresh; i++) {
                             total1=total1+i*hist[i];
                             number1=number1+hist[i];
                      }
                      for (i=thresh; i<256; i++){
                             total2=total2+i*hist[i];
                             number2=number2+hist[i];
                      }
                      if (number1 !=0){
                             avg1=total1/number1;
                      }else {
                             avg1=thresh/2;
                      }
                      if (number2 !=0){
                             avg2=total2/number2;
```

```
                }else {
                        avg2=(255-thresh)/2;
                }
              thresh=(avg1+avg2)/2;
                if (avg1==old1 && avg2==old2){
                        fprintf(stderr, "avg1 = %d, avg2 = %d\n", avg1, avg2);
                        break;
                }else{
                        old1=avg1;
                        old2=avg2;
                }
        }

            if (VFLAG)
                 fprintf(stderr, "thresh = %d\n",thresh);



        /* apply the threshold */
        for (i = im.ylo; i <= im.yhi; i++) {
            for (j = im.xlo; j <= im.xhi; j++) {
                if (im.u[i][j] >= thresh) im.u[i][j] = 255;
                else                      im.u[i][j] = 0;
            }
        }

/************** End of the Application specific section **************/

        VXresetimage(&im); /* free the im image structure  */
        VXptr = VXptr->next; /* move to the next image */
      } /* end of every image section */
      VXwriteframe(VXout,VXlist); /* write frame */
      VXdellist(VXlist);          /* delete the frame */
   } /* end of every frame section */
   VXclose(VXin);  /* close files */
   VXclose(VXout);
   exit(0);
}

/*****************************************************************/
/* VisX4 program watershed                                       */
/* Performs watershed transform                                  */
/*                                                               */
/*****************************************************************/
```

```
#include "VisXV4.h"          /* VisX structure include file     */
#include "Vutil.h"           /* VisX utility header files       */
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

extern char *VisXhist;

char * pname = "v3tpl";

VisXfile_t *VXin,*VXin2,      /* input file structure            */
          *VXout;           /* output file structure           */
VisXelem_t *VXlist,**VXpt;   /* VisX data structure             */
VisXelem_t *mlist;           /* VisX data structure             */

VisX3dim_t  sim;             /* source image structure          */
VisX3dim_t  tim;             /* temporary image structure       */
VisX3dim_t  tim2;            /* temporary image structure       */
VisX3dim_t  rim;             /* result image structure          */
VisX3dim_t  mim;             /* mask image structure            */
VisXiinfo_t imginfo;
float xres,yres,zres,ri,rs;

void VX3frameset(VisX3dim_t *is, VisX3dim_t *ir);
//cclabel for regional minima
int label=1;
void setlabel(int i,int j,int k, int n);

VXparam_t par[] = {
  {"if=",    0,   "input file"},
  {"ig=",    0,   "binary marker file"},
  {"of=",    0,   "output file"},
  {"th=",    0,   "threshold value (default: 800)"},
  {"-v",     0,   "verbose flag"},
  { 0,       0,   0},
};

/* Command line parameters are accessed in code by vars below */
#define  IVAL    par[0].val
#define  MVAL    par[1].val
#define  OVAL    par[2].val
#define  TVAL    par[3].val
#define  VFLAG   par[4].val
```

```
int
main(argc, argv)
int argc;
char *argv[];
{
 int i,j,k;
 int xmin,xmax,ymin,ymax,zmin,zmax;
 int thresh;

 VisXelem_t *vptr = NULL, *mptr = NULL;

 VXparse(&argc, &argv, par);    /* parse the command line      */

 VXin  = VXopen(IVAL, 0);       /* open input file             */
 VXout = VXopen(OVAL, 1);       /* open the output file        */

 VXlist = VXread(VXin);         /* read input file             */
 VXgetresinfo( &imginfo);
 VXgetrescale( &imginfo);
 xres = imginfo.xres;
 yres = imginfo.yres;
 zres = imginfo.zres;
 ri = imginfo.ri;
 rs = imginfo.rs;
 if( VFLAG) {
   fprintf(stderr, "img res = %f x %f x %f ri %f rs %f\n",
              imginfo.xres, imginfo.yres, imginfo.zres, imginfo.ri, imginfo.rs);
 }


 if (TVAL ) {
   thresh = atoi(TVAL);
 } else {
   thresh = 800;
 }

 if(VXNIL == (vptr = VXfind(VXlist, VX_PBYTE))){
   fprintf(stderr, "%s: no acceptable input image found, exiting.\n",pname);
   exit(1);
 }

 /* Initialize input image structure */
```

```
VXset3dim(&sim, vptr, VXin);
if(sim.chan != 1){
  fprintf(stderr, "%s: Multi-channel images are not supported.\n",pname);
  exit(1);
}

//fprintf(stderr,"check\n");

/* Apply mask if mask image is specified*/
if ( MVAL ) {
  if ( VFLAG ) {
    fprintf(stderr, "%s: Mask file specified, applying mask...\n",pname);
  }
  VXin2 = VXopen(MVAL, 0);       /* open mask file              */

  /* Read mask file */
  mlist = VXread(VXin2);
  if (VXNIL == (mptr = VXfind(mlist,VX_PBYTE)) ) {
    fprintf(stderr, "%s: Invalid format for mask file.\n",pname);
    exit(1);
  }

  VXset3dim(&mim, mptr, VXin2);

  /* Check if image and mask have same bounding box, warn if not
     Note that this is not a problem for this program, so we don't
     do anything.                                          */
  if ( (sim.xlo != mim.xlo) || (sim.xhi != mim.xhi) ||
       (sim.ylo != mim.ylo) || (sim.yhi != mim.yhi) ||
       (sim.zlo != mim.zlo) || (sim.zhi != mim.zhi) ) {
    fprintf(stderr, "%s: bounding boxes do not match.\n",pname);
  }

  /* Determine what regions overlap */
  xmin = (sim.xlo>mim.xlo) ? sim.xlo : mim.xlo;
  ymin = (sim.ylo>mim.ylo) ? sim.ylo : mim.ylo;
  zmin = (sim.zlo>mim.zlo) ? sim.zlo : mim.zlo;
  xmax = (sim.xhi<mim.xhi) ? sim.xhi : mim.xhi;
  ymax = (sim.yhi<mim.yhi) ? sim.yhi : mim.yhi;
  zmax = (sim.zhi<mim.zhi) ? sim.zhi : mim.zhi;
}
//no mask
else {
  VXembed3dim(&mim,&sim,1,1,1,1,1,1);
```

```
  for (k = sim.zlo; k <= sim.zhi; k++) {
          for (j = sim.ylo; j <= sim.yhi; j++) {
              for (i = sim.xlo; i <= sim.xhi; i++) {
                    mim.u[k][j][i] = 255;
              }
      }
  }
}
//fprintf(stderr,"checkmask\n");
/* Create result image structure */

VXmake3dim(&rim, VX_PBYTE, sim.bbx, sim.chan);

/*****************************************************/
/***************WATERSHED ALGORITHM******************/
VXembed3dim(&tim,&sim,1,1,1,1,1,1);
VXembed3dim(&tim2,&sim,1,1,1,1,1,1);
//fprintf(stderr,"checkbegin\n");
//reset for labelling
for (k = sim.zlo; k <= sim.zhi; k++) {
  for (j = sim.ylo; j <= sim.yhi; j++) {
    for (i = sim.xlo; i <= sim.xhi; i++) {
        sim.u[k][j][i] = 0;
            tim2.u[k][j][i] = 0;
    }
  }
}
//fprintf(stderr,"check\n");
/*
const int xsize = xmax-xmin+1;
const int ysize = ymax-ymin+1;
const int zsize = zmax-ymin+1;
fprintf(stderr,"check\n");
float gradim[zsize][ysize][xsize]; //gradient array
fprintf(stderr,"checkgrad\n");
*/
float tolerance = 0.01;
float gx, gy, gradient;
/*******Regional Minima Calculation*********************/
for (k = sim.zlo; k <= sim.zhi; k++) {
  for (j = sim.ylo; j <= sim.yhi; j++) {
    for (i = sim.xlo; i <= sim.xhi; i++) {
          gx = -tim.u[k][j-1][i-1]+-2*tim.u[k][j][i-1]+-tim.u[k][j+1][i-1]
          +tim.u[k][j-1][i+1]+2*tim.u[k][j][i+1]+tim.u[k][j+1][i+1];
```

```
      gy = -tim.u[k][j-1][i-1]+-2*tim.u[k][j-1][i]+-tim.u[k][j-1][i+1]
          +tim.u[k][j+1][i-1]+2*tim.u[k][j+1][i]+tim.u[k][j+1][i+1];
      //gradim[k][j][i] = sqrt(gx*gx + gy*gy);
      gradient = sqrt(gx*gx + gy*gy);
          //fprintf(stderr,"%f\n",gradient);
          //rim.u[k][j][i] = (int) gradient;
          /*
      if (gradim[k][j][i]<tolerance && mim.u[k][j][i]==255){
                  sim.u[k][j][i]=255;
      }*/
          if (gradient<tolerance && mim.u[k][j][i]==255){
                  sim.u[k][j][i]=255;
      }
      else {
          sim.u[k][j][i]=0;
      }
    }
  }
}
//fprintf(stderr,"checkrmin\n");
//tim original data
//sim where regional minima exist
/************Label Regional Minima*********************/
for (k = sim.zlo; k <= sim.zhi; k++) {
  for (j = sim.ylo; j <= sim.yhi; j++) {
    for (i = sim.xlo; i <= sim.xhi; i++) {
          if (sim.u[k][j][i]!=0 && tim2.u[k][j][i]==0){
                  setlabel(i,j,k,label);
                  label=label+1;
              }
    }
   }
   label=1;
}
//fprintf(stderr,"checklabel\n");
//tim2 labeled regional minima
/***********Watershed Flooding*******************************/
int queuex[10000];//contains x positon of pixel
int queuey[10000];//contains y position of pixel
int queuep[10000];//contains pixel value (min prioiritized)

int count = 0;
//fprintf(stderr,"check\n");
//initial queue
```

```
for (k = sim.zlo; k <= sim.zhi; k++) {
  fprintf(stderr,"check k=%d\n",k);
  for (j = sim.ylo; j <= sim.yhi; j++) {
    //fprintf(stderr,"check j=%d\n",j);
    //fprintf(stderr,"count %d\n",count);
    for (i = sim.xlo; i <= sim.xhi; i++) {
                    //fprintf(stderr,"check i=%d\n",i);
                    if (tim2.u[k][j][i] != 0){//labeled
                    //add neighbors (and their positions) to queue
                      if(tim2.u[k][j-1][i]==0){
                            tim2.u[k][j-1][i] = tim2.u[k][j][i];
                            queuex[count] = i;
                            queuey[count] = j-1;
                            queuep[count] = tim.u[k][j-1][i];
                            count=count+1;
                      }
                      if(tim2.u[k][j+1][i]==0){
                            tim2.u[k][j+1][i] = tim2.u[k][j][i];
                            queuex[count] = i;
                            queuey[count] = j+1;
                            queuep[count] = tim.u[k][j+1][i];
                            count=count+1;
                      }
                      if(tim2.u[k][j][i-1]==0){
                            tim2.u[k][j][i-1] = tim2.u[k][j][i];
                            queuex[count] = i-1;
                            queuey[count] = j;
                            queuep[count] = tim.u[k][j][i-1];
                            count=count+1;
                      }
                      if(tim2.u[k][j][i+1]==0){
                            tim2.u[k][j][i+1] = tim2.u[k][j][i];
                            queuex[count] = i+1;
                            queuey[count] = j;
                            queuep[count] = tim.u[k][j][i+1];
                            count=count+1;
                      }
                    }
    }
  }
  fprintf(stderr,"checkinitial\n");
  //selection sort in ascending order
  int iMin,tempx,tempy,tempp;
  int a,b;
```

```
for (a = 0; a < count; a++){//last count++ means count = sizeof(queue)+1
	iMin = a;
	for (b = a+1; b < count; b++){
		if (queuep[b] < queuep[iMin]) iMin = b;
	}
	if (iMin != a) {
		tempx = queuex[a];
		tempy = queuey[a];
		tempp = queuep[a];
		queuep[a] = queuep[iMin];
		queuex[a] = queuex[iMin];
		queuey[a] = queuey[iMin];
		queuep[iMin] = tempp;
		queuex[iMin] = tempx;
		queuey[iMin] = tempy;
	}
}
/*for (a = 0; a < count; a++){
	fprintf(stderr,"queuep%d\n",queuep[a]);
}*/
while (count>0){
	fprintf(stderr,"checkcount%d\n",count);
	//add neighbors (and their positions) to queue
	if(tim2.u[k][queuey[0]-1][queuex[0]]=0){
		//fprintf(stderr,"check1\n");
		tim2.u[k][queuey[0]-1][queuex[0]] = tim2.u[k][queuey[0]][queuex[0]];
		queuex[count] = i;
		queuey[count] = j-1;
		queuep[count] = tim.u[k][j-1][i];
		count=count+1;
	}
	if(tim2.u[k][queuey[0]+1][queuex[0]]==0){
		//fprintf(stderr,"check2\n");
		tim2.u[k][queuey[0]+1][queuex[0]] = tim2.u[k][queuey[0]][queuex[0]];
		queuex[count] = i;
		queuey[count] = j+1;
		queuep[count] = tim.u[k][j+1][i];
		count=count+1;
	}
	if(tim2.u[k][queuey[0]][queuex[0]-1]==0){
		//fprintf(stderr,"check3\n");
		tim2.u[k][queuey[0]][queuex[0]-1] = tim2.u[k][queuey[0]][queuex[0]];
		queuex[count] = i-1;
		queuey[count] = j;
```

```
            queuep[count] = tim.u[k][j][i-1];
            count=count+1;
        }
        if(tim2.u[k][queuey[0]][queuex[0]+1]==0){
            //fprintf(stderr,"check4\n");
            tim2.u[k][queuey[0]][queuex[0]+1] = tim2.u[k][queuey[0]][queuex[0]];
            queuex[count] = i+1;
            queuey[count] = j;
            queuep[count] = tim.u[k][j][i+1];
            count=count+1;
        }
        //fprintf(stderr,"checkadd\n");
        //remove first pixel from queue
        for (a = 0; a < count-1; a++){
            queuex[a] = queuex[a+1];
            queuey[a] = queuey[a+1];
            queuep[a] = queuep[a+1];
        }
        count = count - 1;
        //fprintf(stderr,"checkremove\n");
        //resort queue

        for (a = 0; a < count; a++){//last count++ means count = sizeof(queue)+1
            iMin = a;
            for (b = a+1; b < count; b++){
                if (queuep[b] < queuep[iMin]) iMin = b;
            }
            if (iMin != a) {
                tempx = queuex[a];
                tempy = queuey[a];//fprintf(stderr,"count %d\n",count);
                tempp = queuep[a];
                queuep[a] = queuep[iMin];
                queuex[a] = queuex[iMin];
                queuey[a] = queuey[iMin];
                queuep[iMin] = tempp;
                queuex[iMin] = tempx;
                queuey[iMin] = tempy;
            }
         }
         //fprintf(stderr,"checksort\n");

}
//fprintf(stderr,"count %d\n",count);
//end of kth slice
```

```
 }
 fprintf(stderr,"checkwend\n");
 for (k = sim.zlo; k <= sim.zhi; k++) {
   for (j = sim.ylo; j <= sim.yhi; j++) {
     for (i = sim.xlo; i <= sim.xhi; i++) {
          rim.u[k][j][i] = (tim2.u[k][j][i]==0) ? 255 : 0;
     }
   }
 }
 /******************END OF ALGORITHM*********************/
 /***************************************************/

 VX3frameset(&sim,&rim);

 VXwrite(VXout, rim.list);     /* write data                */
 VXclose(VXin);               /* close files               */
 VXclose(VXout);

 exit(0);
}


void setlabel(int i,int j, int k, int n){
          tim2.u[k][j][i]=n;

          if (sim.u[k][j][i+1]!=0 && tim2.u[k][j][i+1]==0){
                    setlabel(i+1,j,k,n);
                  //fprintf(stderr, "check1\n");
          }
          if (sim.u[k][j][i-1]!=0 && tim2.u[k][j][i-1]==0){
                    setlabel(i-1,j,k,n);
                    //fprintf(stderr, "check2\n");
          }
          if (sim.u[k][j-1][i]!=0 && tim2.u[k][j-1][i]==0){
                    setlabel(i,j-1,k,n);
                    //fprintf(stderr, "check3\n");
          }
          if (sim.u[k][j+1][i]!=0 && tim2.u[k][j+1][i]==0){
                    setlabel(i,j+1,k,n);
                    //fprintf(stderr, "check4\n");
          }

}
```

```
void VX3frameset(VisX3dim_t *is, VisX3dim_t *ir) {
/* VX3frameset: inserts frame markers in ir in the same location
  as in is

  This function assumes that both 3D images have the same number
  of images. If not, it will print an error and quit.
*/
   VisXelem_t *src, *dest;
   VisXelem_t *fptr,*sptr;    /* ptrs into src list */
   VisXelem_t *dptr;          /* ptrs into dst list */

   /* ensure at the beginning of each list */
   src=VXfirst(is->list);
   dest=VXfirst(ir->list);

   if ( VXNIL == (fptr = VXfind(src, VX_FRAME)) ) {
      /* No frames, don't do anything */
      return;
   }
   /* start searching after the first frame marker */
   sptr = src;
   dptr = dest;
   while(VXNIL != (sptr = VXfind(sptr, VX_BBX)) ) {
      if (VXNIL == (dptr = VXfind(dptr, VX_BBX)) ) {
         fprintf(stderr, "Error: Image count not equal!\n");
         exit(1);
      }
      fptr = VXbfind(sptr, VX_FRAME);
      /* Go back one element from BBX to add frame marker */
      dptr = VXaddelem(dptr->prev,VX_FRAME,"",fptr->size);

      fptr = VXfind(fptr, VX_EFRAME);
      /* Set pointer to the image (bbx, then image) */
      /* TODO: Implement variable number of element skipping */
      dptr = dptr->next->next;
      dptr = VXaddelem(dptr, VX_EFRAME, "", fptr->size);
      sptr = sptr->next;
   }
   if (VXNIL != VXfind(dptr, VX_BBX)) {
      fprintf(stderr, "Error: Image count not equal at end!\n");
      exit(1);
   }
```

```
}

/*****************************************************************/
/* VisX4 program ssra                                            */
/* Removes small segments                                        */
/*                                                               */
/*****************************************************************/
#include "VisXV4.h"          /* VisX structure include file     */
#include "Vutil.h"           /* VisX utility header files       */

extern char *VisXhist;

char * pname = "v3tpl";

VisXfile_t *VXin,*VXin2,      /* input file structure           */
         *VXout;           /* output file structure           */
VisXelem_t *VXlist,**VXpt;   /* VisX data structure             */
VisXelem_t *mlist;           /* VisX data structure             */

VisX3dim_t  sim;             /* source image structure          */
VisX3dim_t  rim;             /* result image structure          */
VisX3dim_t  mim;             /* mask image structure            */

VisX3dim_t exm;                      /* external image */
VisX3dim_t tm;               /* temp image     */

VisXiinfo_t imginfo;
float xres,yres,zres,ri,rs;

void VX3frameset(VisX3dim_t *is, VisX3dim_t *ir);

VXparam_t par[] = {
  {"if=",    0,   "input file"},
  {"ig=",    0,   "binary mask file"},
  {"of=",    0,   "output file"},
  {"th=",    0,   "threshold value (default: 800)"},
  {"-v",     0,   "verbose flag"},
  { 0,       0,   0},
};

/* Command line parameters are accessed in code by vars below */
#define  IVAL    par[0].val
#define  MVAL    par[1].val
#define  OVAL    par[2].val
```

```
#define  TVAL    par[3].val
#define  VFLAG   par[4].val



const int thresh; // threshold for connected components
int count = 0; // count for the SSRA function
int removeQueuex[thresh]; // queue for the small segment removal
int removeQueuey[thresh];

void SSRA(int i, int j, int k); // initialize functions
void SSRAr();
int i,j,k;



int
main(argc, argv)
int argc;
char *argv[];
{
 int xmin,xmax,ymin,ymax,zmin,zmax;
 int thresh;

 VisXelem_t *vptr = NULL, *mptr = NULL;

 VXparse(&argc, &argv, par);    /* parse the command line      */

 VXin  = VXopen(IVAL, 0);       /* open input file             */
 VXout = VXopen(OVAL, 1);       /* open the output file        */

 VXlist = VXread(VXin);         /* read input file             */
 VXgetresinfo( &imginfo);
 VXgetrescale( &imginfo);
 xres = imginfo.xres;
 yres = imginfo.yres;
 zres = imginfo.zres;
 ri = imginfo.ri;
 rs = imginfo.rs;
 if( VFLAG) {
   fprintf(stderr, "img res = %f x %f x %f ri %f rs %f\n",
              imginfo.xres, imginfo.yres, imginfo.zres, imginfo.ri, imginfo.rs);
 }
```

```
if (TVAL ) {
  thresh = atoi(TVAL);
} else {
  thresh = 800;
}

if(VXNIL == (vptr = VXfind(VXlist, VX_PBYTE))){
  fprintf(stderr, "%s: no acceptable input image found, exiting.\n",pname);
  exit(1);
}

/* Initialize input image structure */
VXset3dim(&sim, vptr, VXin);
if(sim.chan != 1){
  fprintf(stderr, "%s: Multi-channel images are not supported.\n",pname);
  exit(1);
}

/* Apply mask if mask image is specified*/
if ( MVAL ) {
  if ( VFLAG ) {
    fprintf(stderr, "%s: Mask file specified, applying mask...\n",pname);
  }
  VXin2 = VXopen(MVAL, 0);       /* open mask file              */

  /* Read mask file */
  mlist = VXread(VXin2);
  if (VXNIL == (mptr = VXfind(mlist,VX_PBYTE)) ) {
    fprintf(stderr, "%s: Invalid format for mask file.\n",pname);
    exit(1);
  }

  VXset3dim(&mim, mptr, VXin2);

  /* Check if image and mask have same bounding box, warn if not
     Note that this is not a problem for this program, so we don't
     do anything.                                          */
  if ( (sim.xlo != mim.xlo) || (sim.xhi != mim.xhi) ||
       (sim.ylo != mim.ylo) || (sim.yhi != mim.yhi) ||
       (sim.zlo != mim.zlo) || (sim.zhi != mim.zhi) ) {
    fprintf(stderr, "%s: bounding boxes do not match.\n",pname);
  }
```

```
  /* Determine what regions overlap */
   xmin = (sim.xlo>mim.xlo) ? sim.xlo : mim.xlo;
  ymin = (sim.ylo>mim.ylo) ? sim.ylo : mim.ylo;
  zmin = (sim.zlo>mim.zlo) ? sim.zlo : mim.zlo;
  xmax = (sim.xhi<mim.xhi) ? sim.xhi : mim.xhi;
  ymax = (sim.yhi<mim.yhi) ? sim.yhi : mim.yhi;
  zmax = (sim.zhi<mim.zhi) ? sim.zhi : mim.zhi;

}
//no mask
else {
  for (k = sim.zlo; k <= sim.zhi; k++) {
          for (j = sim.ylo; j <= sim.yhi; j++) {
              for (i = sim.xlo; i <= sim.xhi; i++) {
                    mim.u[k][j][i] = 255;
              }
      }
  }
}
/* Create result image structure */
VXmake3dim(&rim, VX_PBYTE, sim.bbx, sim.chan);

/*********************************************************/
/******************ALGORITHM: EDIT THIS******************/



     /********************** Small Segment Removal Algorithm **********************/
for (k = sim.zlo; k <= sim.zhi; k++){
          for (i = sim.ylo; i<= sim.yhi; i++) {
                    for (j = sim.xlo; j <= sim.xhi; j++) {
                              if (sim.u[k][i][j] != 0) { SSRA(i,j,k); }
                              if (count <= thresh) { SSRAr();          }
                              count = 0;
                    }
          }


}

/**********************END OF ALGORITHM*******************/
/*********************************************************/
```

```
 VX3frameset(&sim,&rim);

 VXwrite(VXout, rim.list);     /* write data                */
 VXclose(VXin);               /* close files               */
 VXclose(VXout);

 exit(0);
}

/* SSRA: recursively counts connected components for size */
void SSRA(int i, int j, int k) {
        int breakf = 0;
        if (count < thresh) {
                removeQueuex[count] = i;
                removeQueuey[count] = j;
        }
        else {
                memset(removeQueuex, 0, thresh);
                memset(removeQueuey, 0, thresh);
                breakf=1;
        }

        if (breakf==0){
        if (tm.u[k][i][j+1] == 255 && sim.u[k][i][j+1] == 255) {
                count = count + 1;
                SSRA(i, j+1,k);
        }
        if (tm.u[k][i][j-1] == 255 && sim.u[k][i][j-1] == 255) {
                count = count + 1;
                SSRA(i,j-1,k);
        }
        if (tm.u[k][i+1][j] == 255 && sim.u[k][i+1][j] == 255) {
                count = count + 1;
                SSRA(i+1,j,k);
        }
        if (tm.u[k][i-1][j] == 255 && sim.u[k][i-1][j] == 255) {
                count = count + 1;
                SSRA(i-1,j,k);
        }
        }
}

/* SSRA: removes the connected components that are under threshold */
void SSRAr() {
```

```
        for (i = 0; i < thresh; i++) {
                sim.u[k][removeQueuex[i]][removeQueuey[i]] = 0;
        }
}
```

```
void VX3frameset(VisX3dim_t *is, VisX3dim_t *ir) {
/* VX3frameset: inserts frame markers in ir in the same location
  as in is

  This function assumes that both 3D images have the same number
  of images. If not, it will print an error and quit.
*/
   VisXelem_t *src, *dest;
   VisXelem_t *fptr,*sptr;    /* ptrs into src list */
   VisXelem_t *dptr;          /* ptrs into dst list */

   /* ensure at the beginning of each list */
   src=VXfirst(is->list);
   dest=VXfirst(ir->list);

   if ( VXNIL == (fptr = VXfind(src, VX_FRAME)) ) {
      /* No frames, don't do anything */
      return;
   }
   /* start searching after the first frame marker */
   sptr = src;
   dptr = dest;
   while(VXNIL != (sptr = VXfind(sptr, VX_BBX)) ) {
      if (VXNIL == (dptr = VXfind(dptr, VX_BBX)) ) {
         fprintf(stderr, "Error: Image count not equal!\n");
         exit(1);
      }
      fptr = VXbfind(sptr, VX_FRAME);
      /* Go back one element from BBX to add frame marker */
      dptr = VXaddelem(dptr->prev,VX_FRAME,"",fptr->size);

      fptr = VXfind(fptr, VX_EFRAME);
      /* Set pointer to the image (bbx, then image) */
      /* TODO: Implement variable number of element skipping */
      dptr = dptr->next->next;
```

```
      dptr = VXaddelem(dptr, VX_EFRAME, "", fptr->size);
      sptr = sptr->next;
   }
   if (VXNIL != VXfind(dptr, VX_BBX)) {
      fprintf(stderr, "Error: Image count not equal at end!\n");
      exit(1);
   }
}

/*****************************************************************/
/* VisX4 program evaluation                                      */
/* RMSD evaluation                                               */
/*                                                               */
/*****************************************************************/
#include "VisXV4.h"          /* VisX structure include file     */
#include "Vutil.h"           /* VisX utility header files       */

extern char *VisXhist;

char * pname = "v3tpl";

VisXfile_t *VXin,*VXin2,      /* input file structure            */
         *VXout;           /* output file structure           */
VisXelem_t *VXlist,**VXpt;   /* VisX data structure             */
VisXelem_t *mlist;           /* VisX data structure             */

VisX3dim_t  sim;             /* source image structure          */
VisX3dim_t  rim;             /* result image structure          */
VisX3dim_t  mim;             /* mask image structure            */
VisXiinfo_t imginfo;
float xres,yres,zres,ri,rs;

void VX3frameset(VisX3dim_t *is, VisX3dim_t *ir);

VXparam_t par[] = {
  {"if=",    0,   "input file"},
  {"ig=",    0,   "ground truth file"},
  {"of=",    0,   "output file"},
  {"s=",     0,   "fissure size (default: 10000)"},
  {"-v",     0,   "verbose flag"},
  { 0,       0,   0},
};

/* Command line parameters are accessed in code by vars below */
```

```
#define  IVAL    par[0].val
#define  MVAL    par[1].val
#define  OVAL    par[2].val
#define  SVAL    par[3].val
#define  VFLAG   par[4].val

int
main(argc, argv)
int argc;
char *argv[];
{
 int i,j,k;
 int xmin,xmax,ymin,ymax,zmin,zmax;

 VisXelem_t *vptr = NULL, *mptr = NULL;

 VXparse(&argc, &argv, par);    /* parse the command line      */

 VXin  = VXopen(IVAL, 0);       /* open input file             */
 VXout = VXopen(OVAL, 1);       /* open the output file        */

 VXlist = VXread(VXin);         /* read input file             */
 VXgetresinfo( &imginfo);
 VXgetrescale( &imginfo);
 xres = imginfo.xres;
 yres = imginfo.yres;
 zres = imginfo.zres;
 ri = imginfo.ri;
 rs = imginfo.rs;
 if( VFLAG) {
   fprintf(stderr, "img res = %f x %f x %f ri %f rs %f\n",
              imginfo.xres, imginfo.yres, imginfo.zres, imginfo.ri, imginfo.rs);
 }


 if(VXNIL == (vptr = VXfind(VXlist, VX_PBYTE))){
   fprintf(stderr, "%s: no acceptable input image found, exiting.\n",pname);
   exit(1);
 }

 /* Initialize input image structure */
 VXset3dim(&sim, vptr, VXin);
 if(sim.chan != 1){
   fprintf(stderr, "%s: Multi-channel images are not supported.\n",pname);
```

```
  exit(1);
}

/* Apply mask if mask image is specified*/
if ( MVAL ) {
  if ( VFLAG ) {
    fprintf(stderr, "%s: Mask file specified, applying mask...\n",pname);
  }
  VXin2 = VXopen(MVAL, 0);       /* open mask file              */

  /* Read mask file */
  mlist = VXread(VXin2);
  if (VXNIL == (mptr = VXfind(mlist,VX_PBYTE)) ) {
    fprintf(stderr, "%s: Invalid format for mask file.\n",pname);
    exit(1);
  }

  VXset3dim(&mim, mptr, VXin2);

  /* Check if image and mask have same bounding box, warn if not
     Note that this is not a problem for this program, so we don't
     do anything.                                          */
  if ( (sim.xlo != mim.xlo) || (sim.xhi != mim.xhi) ||
       (sim.ylo != mim.ylo) || (sim.yhi != mim.yhi) ||
       (sim.zlo != mim.zlo) || (sim.zhi != mim.zhi) ) {
    fprintf(stderr, "%s: bounding boxes do not match.\n",pname);
  }

  /* Determine what regions overlap */
  xmin = (sim.xlo>mim.xlo) ? sim.xlo : mim.xlo;
  ymin = (sim.ylo>mim.ylo) ? sim.ylo : mim.ylo;
  zmin = (sim.zlo>mim.zlo) ? sim.zlo : mim.zlo;
  xmax = (sim.xhi<mim.xhi) ? sim.xhi : mim.xhi;
  ymax = (sim.yhi<mim.yhi) ? sim.yhi : mim.yhi;
  zmax = (sim.zhi<mim.zhi) ? sim.zhi : mim.zhi;

}
// no mask
else {
  for (k = sim.zlo; k <= sim.zhi; k++) {
          for (j = sim.ylo; j <= sim.yhi; j++) {
              for (i = sim.xlo; i <= sim.xhi; i++) {
                    mim.u[k][j][i] = 255;
              }
```

```
    }
  }
}
/* Create result image structure */
VXmake3dim(&rim, VX_PBYTE, sim.bbx, sim.chan);

/* set history to history of source image file */
/* this is not necessary since the result image was opened
   before the mask image
   VXhistset(VXin->list, 1);
*/

/************************************************/
/********EVAL ALGORITHM*****************************/
int size;
if (SVAL ) {
  size = atoi(SVAL);
} else {
  size = 10000;
}
int basex[size];
int basey[size];
int testx[size];
int testy[size];
int count = 0;
int counti = 0;
int xdiff = 0;
int ydiff = 0;
int hyp = 0;
int distTotal = 0;
for (k = sim.zlo; k <= sim.zhi; k++) {
  for (j = sim.ylo; j <= sim.yhi; j++) {
    for (i = sim.xlo; i <= sim.xhi; i++) {
      /* Do processing stuff here */
      if (sim.u[k][j][i] == 255) {
                    basex[count] = i;
                    basey[count] = j;
          }
          if (mim.u[k][j][i] == 255) {
                rim.u[k][j][i] = 255;
                    testx[counti] = i;
                    testy[counti] = j;
          }
      else {
```

```
                    rim.u[k][j][i] = 0;
      }
     }
   }
  }
  for (k = 0; k < size; k++) {
          xdiff = (basex[k] - testx[k])*(basex[k] - testx[k]);
          ydiff = (basey[k] - testy[k])*(basey[k] - testy[k]);
          hyp = sqrt(xdiff*xdiff + ydiff*ydiff);
          distTotal = distTotal + hyp;
  }
  distTotal = distTotal / size;
  printf("RMSD is %d", distTotal);
  /***********************************************/
  VX3frameset(&sim,&rim);

  VXwrite(VXout, rim.list);     /* write data                  */
  VXclose(VXin);               /* close files                 */
  VXclose(VXout);

  exit(0);
}

void VX3frameset(VisX3dim_t *is, VisX3dim_t *ir) {
/* VX3frameset: inserts frame markers in ir in the same location
   as in is

   This function assumes that both 3D images have the same number
   of images. If not, it will print an error and quit.
*/
    VisXelem_t *src, *dest;
    VisXelem_t *fptr,*sptr;    /* ptrs into src list */
    VisXelem_t *dptr;          /* ptrs into dst list */

    /* ensure at the beginning of each list */
    src=VXfirst(is->list);
    dest=VXfirst(ir->list);

    if ( VXNIL == (fptr = VXfind(src, VX_FRAME)) ) {
        /* No frames, don't do anything */
        return;
    }
    /* start searching after the first frame marker */
    sptr = src;
```

```
    dptr = dest;
    while(VXNIL != (sptr = VXfind(sptr, VX_BBX)) ) {
        if (VXNIL == (dptr = VXfind(dptr, VX_BBX)) ) {
            fprintf(stderr, "Error: Image count not equal!\n");
            exit(1);
        }
        fptr = VXbfind(sptr, VX_FRAME);
        /* Go back one element from BBX to add frame marker */
        dptr = VXaddelem(dptr->prev,VX_FRAME,"",fptr->size);

        fptr = VXfind(fptr, VX_EFRAME);
        /* Set pointer to the image (bbx, then image) */
        /* TODO: Implement variable number of element skipping */
        dptr = dptr->next->next;
        dptr = VXaddelem(dptr, VX_EFRAME, "", fptr->size);
        sptr = sptr->next;
    }
    if (VXNIL != VXfind(dptr, VX_BBX)) {
        fprintf(stderr, "Error: Image count not equal at end!\n");
        exit(1);
    }
}
```