ECE 5273
HW 2 Solution

Spring 2022 Dr. Havlicek

Note: This document contains solutions in both Matlab and traditional C.

C Solution:

The program `bthresh2dir.c` given below was used to perform thresholding. For each image, the remainder of the processing was done by the program `hw02Blob.c`, also given below.

For `Suzi1.bin`, the desired "girl" object has gray levels that are less than the background. Therefore, for this image the threshold result is set to LOGIC_ONE for pixels that are *below* threshold. Note that `bthresh2dir` accomodates this requirement if the command line parameter `direction` is equal to zero. The threshold was determined empirically by trial and error. What was needed was a threshold value for which

1. the perimeter of the "girl" object would be covered by a single connected component of value LOGIC_ONE,
2. this single connected component of value LOGIC_ONE would contain more pixels than the sum of the number of pixels contained in all LOGIC_ZERO connected components contained within the perimeter.

As can be seen from the thresholding result given below, the threshold value $\tau = 95$ satisfied these requirements and was used for the `Suzi1.bin` image.

For `ct_scan.bin`, the desired "torso section" object has gray levels that are greater than the background. Therefore, for this image the threshold result is set to LOGIC_ONE for pixels that are *above* threshold. Note that `bthresh2dir` accomodates this requirement if the command line parameter `direction` is equal to one. The threshold was determined empirically by trial and error. The threshold requirements were the same as for `Suzi1.bin`, as articulated above. As can be seen from the thresholding result given below, the threshold value $\tau = 1$ satisfied theses requirements and was used for the `ct_scan` image.

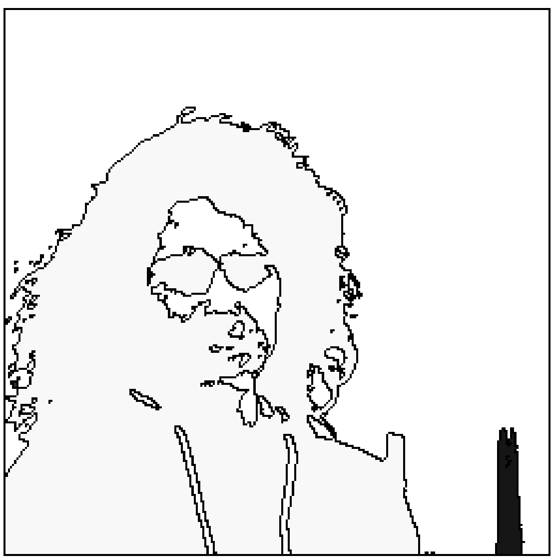
For both input images, most of the gray levels in the image of blob labels (blob colored image) are taken up by very tiny blobs. This results in a poor display of the blob colored image. Therefore, two enhancements were applied. First, a full scale contrast stretch was applied to the blob labels and the stretched labels were then inverted with respect to gray level 255. Second, the primitive approximate contour generation algorithm given in the notes was applied to create contours between the blobs. These contours (or "blob edges") are shown in black in the images below.

- `Suzi1`:

Original `Suzi1` Image

Threshold result, $\tau = 95$

Result after blob coloring



Result after minor region removal

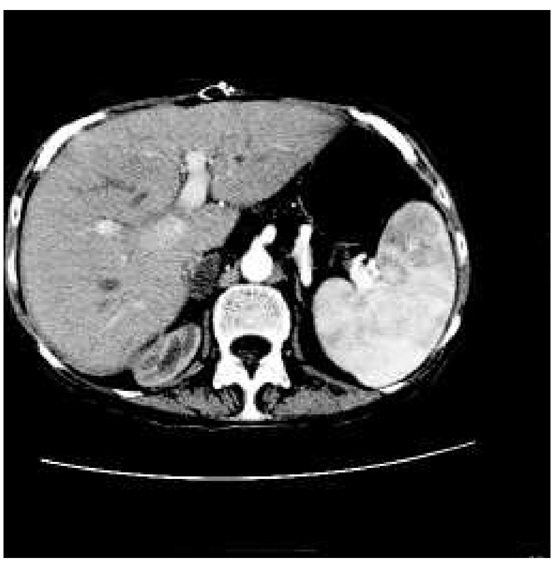
Note: In the "Result after blob coloring" above, I have added a black outline around some of the blobs to enhance the readability.

Final result

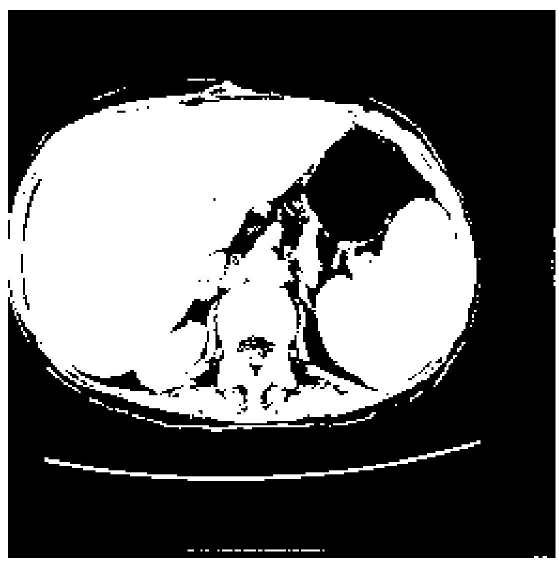
Fused result (extracted object)

- `ct_scan`:

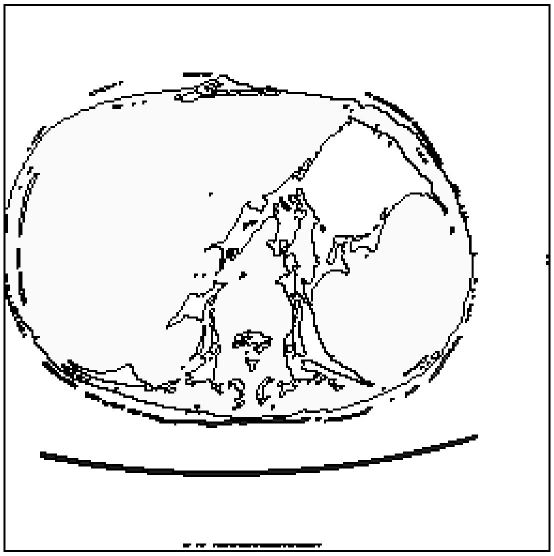
Original `ct_scan` Image



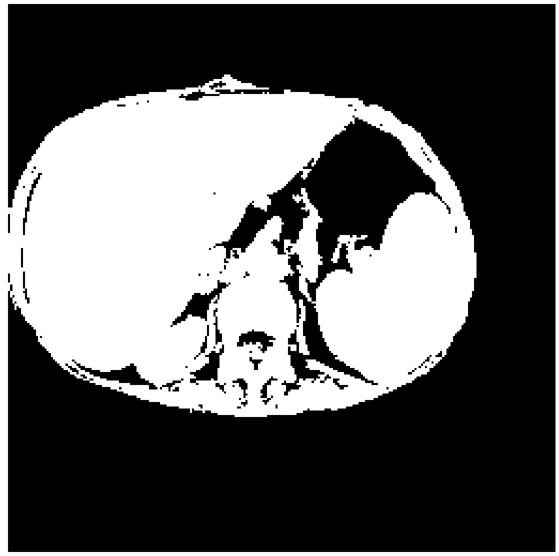
Threshold result, $\tau = 1$



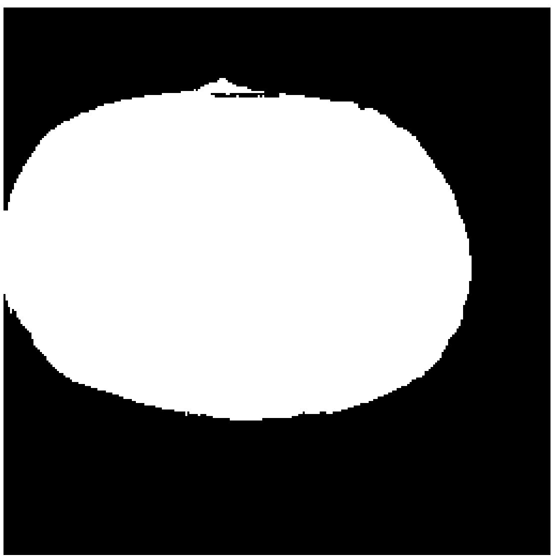
Result after blob coloring



Result after minor region removal



Final result



Fused result (extracted object)

C program listing:

```
/*
 * hw02Blob.c:
 *
 *   - Read in a size x size BYTE image x.
 *   - Perform blob coloring, blob counting, and minor region removal.
 *   - Invert the image and repeat to "fill in holes".
 *   - Invert the result above to obtain the final answer.
 *
 *   This version builds under
 *      gcc version egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)
 *
 * 4/12/2002 jph
 *
 */

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>

#define BYTE unsigned char
#define LOGIC_ONE (0xff)
#define LOGIC_ZERO (0x00)

/*
 * Function Prototypes (forward declarations)
 */
void disk2byte();
void byte2disk();
int  BlobColor();
void BlobCount();
void RemoveMinorRegions();
void InvertBYTEImage();
BYTE *i2bdbx();
BYTE *FuseForDisplay();
void WriteAsciiArrayBYTE();
void WriteAsciiArrayInt();


/*----------------------------------------------------------------------*/
/*   MAIN                                                               */
/*----------------------------------------------------------------------*/

main(argc,argv)

  int     argc;
  char   *argv[];
{

  int      size;                 /* num rows/cols in images */
  char    *InFn;                 /* input filename */
  char    *OrigFn;               /* original image filename */
  char    *BlobFn;               /* Filename for blob labels after coloring */
  char    *BlobFnMinor;          /* Fn for BlobLabels after color & removal */
  char    *OutFn;                /* output filename */
  char    *DisplayFuseFn;        /* Display image Fn: fuse rslt w/ original */
  BYTE    *x;                    /* input image */
  BYTE    *y;                    /* BYTE result image to be reported */
  int      N;                    /* number of pixels = size * size */
  int      i;                    /* counter */
  int     *BlobLabels;           /* array of blob colors */
  int     *BlobCounts;           /* blob counting results */
  int      MaxLabel;             /* biggest value in BlobLabels */

 /*
```

```
 * Check for proper invocation, parse args
 */
 if (argc != 8) {
   printf("\n%s: Blob coloring w/ minor region removal for hw02.",argv[0]);
   printf(
       "\nUsage: %s size InFn OrigFn BlobFn BlobFnMinor OutFn DsplyFuseFn\n",
          argv[0]);
   exit(0);
 }
 size = atoi(argv[1]);
 N = size * size;
 InFn = argv[2];
 OrigFn = argv[3];
 BlobFn = argv[4];
 BlobFnMinor = argv[5];
 OutFn = argv[6];
 DisplayFuseFn = argv[7];

/*
 * Allocate image arrays
 */
 if ((x = (BYTE *)malloc(size*size*sizeof(BYTE))) == NULL) {
   printf("\n%s: free store exhausted.\n",argv[0]);
   exit(-1);
 }
 if ((BlobLabels = (int *)malloc(size*size*sizeof(int))) == NULL) {
   printf("\n%s: free store exhausted.\n",argv[0]);
   exit(-1);
 }

/*
 * Read input image and verify that it is a binary image
 */
 disk2byte(x,size,size,InFn);
 for (i=0; i < N; i++) {
   if ((x[i] != LOGIC_ONE) && (x[i] != LOGIC_ZERO)) {
     printf("\n%s: input image must be binary; value %d is illegal!\n\n",
            argv[0],x[i]);
     exit(-1);
   }
 }

/*
 * Perform Blob Coloring, Blob Counting, and minor region removal
 */
 MaxLabel = BlobColor(x,size,BlobLabels);
 y = i2bdbx(BlobLabels,N,size);
 byte2disk(y,size,size,BlobFn);
 free(y);
 if ((BlobCounts = (int *)malloc((MaxLabel+1)*sizeof(int))) == NULL) {
   printf("\n%s: free store exhausted.\n",argv[0]);
   exit(-1);
 }
 BlobCount(size,BlobLabels,MaxLabel,BlobCounts);
 RemoveMinorRegions(x,size,BlobLabels,MaxLabel,BlobCounts);
 byte2disk(x,size,size,BlobFnMinor);
 free(BlobCounts);

/*
 * Invert the image and repeat blob coloring and minor region removal
 */
 InvertBYTEImage(x,N);
 MaxLabel = BlobColor(x,size,BlobLabels);
 if ((BlobCounts = (int *)malloc((MaxLabel+1)*sizeof(int))) == NULL) {
   printf("\n%s: free store exhausted.\n",argv[0]);
   exit(-1);
 }
 BlobCount(size,BlobLabels,MaxLabel,BlobCounts);
```

```
  RemoveMinorRegions(x,size,BlobLabels,MaxLabel,BlobCounts);

 /*
  * Invert the image to obtain the final result
  */
  InvertBYTEImage(x,N);

 /*
  * Write the output image
  */
  byte2disk(x,size,size,OutFn);

 /*
  * Make a special "fused" image that combines the original image with the
  *  blob coloring result.  This image is just for fancy display.
  */
  y = FuseForDisplay(x,OrigFn,size,N,DisplayFuseFn);
  byte2disk(y,size,size,DisplayFuseFn);
  free(y);


  return;
} /*----------------  Main  ---------------------------------------------*/


/*----------------------------------------------------------------------
 * BlobColor
 *
 *   Function performs connected components labeling (blob coloring) on
 *   a square binary image.  The largest label assigned by the algorithm
 *   is returned.
 *
 *
 *  jph 12 April 2002
 *
 ----------------------------------------------------------------------*/

int BlobColor(x,size,BlobLabels)

  BYTE    *x;             /* input image */
  int      size;          /* row/col dimension of image x */
  int     *BlobLabels;    /* computed array of blob labels (colors) */

{

  BYTE    *xCopy;         /* copy of x with one extra row and one extra col */
  int      N;             /* number of pixels = size * size */
  int      NCopy;         /* num pixels in xCopy = (size+1) * (size+1) */
  int      sizeCopy;      /* row/col dim of xCopy = size + 1 */
  int      i;             /* loop counter */
  int      j;             /* loop counter */
  int      iCopy;         /* cursor into xCopy array */
  int      row;           /* image row counter */
  int      col;           /* image col counter */
  int      rowCopy;       /* row counter for copy image */
  int      colCopy;       /* col counter for copy image */
  int      NextLabel;     /* next available color for a new blob */
  int      KillLabel;     /* label to kill */
  int      KeepLabel;     /* label to keep */

 /*
  * Initialize
  */
  N = size * size;
  sizeCopy = size + 1;
  NCopy = sizeCopy * sizeCopy;
  NextLabel = 1;
```

```
 for (i=0; i < N; i++) {
   BlobLabels[i] = 0;
 }

/*
 * Allocate xCopy - a copy of x with one extra row and col prepended
 */
 if ((xCopy = (BYTE *)malloc(NCopy*sizeof(BYTE))) == NULL) {
   printf("\nBlobColor: free store exhausted.\n");
   exit(-1);
 }

/*
 * Initialize first row and first col of xCopy to logic zero
 */
 for (i=0; i < sizeCopy; i++) {
   xCopy[i] = xCopy[i * sizeCopy] = LOGIC_ZERO;
 }

/*
 * The rest of the xCopy array is just a copy of x
 */
 for (row=0,rowCopy=1; row < size; row++,rowCopy++) {
   for (col=0,colCopy=1; col < size; col++,colCopy++) {
     xCopy[rowCopy*sizeCopy + colCopy] = x[row*size + col];
   }
 }

/*
 * Loop on rows and columns, color blobs
 *  -the xCopy array has a copy of the original binary image x, but with
 *   an extra first row of all zeros and an extra first col of all zeros.
 *  -we can now loop over rows and cols starting at ONE instead of zero in the
 *   xCopy array and we don't have to worry about addressing off the left
 *   side or the top of the array in doing the algorithm on page 2.44 of the
 *   course notes.
 *  -doing this requires us to keep track of two sets of row and col counters.
 *   One set loops over rows/cols in xCopy starting at ONE.  The other set
 *   simultaneoulsy loops over the corresponding points in the BlobLabels
 *   array starting each row/col loop at ZERO.
 */
 for (i=row=0,rowCopy=1; row < size; row++,rowCopy++) {
   for (col=0,colCopy=1; col < size; col++,colCopy++,i++) {
     iCopy = rowCopy * sizeCopy + colCopy;
       if (xCopy[iCopy]) {
         if ((!xCopy[iCopy-1]) && (!xCopy[iCopy-sizeCopy])) {
           /* this is the first case on page 2.44 of the notes */
           BlobLabels[i] = NextLabel++;
         } else {     /* ((!xCopy[iCopy-1]) && (!xCopy[iCopy-sizeCopy)) */
           if ((!xCopy[iCopy-1]) && (xCopy[iCopy-sizeCopy])) {
              BlobLabels[i] =  BlobLabels[i-size];
           } else {    /* ((!xCopy[iCopy-1]) && (xCopy[iCopy-sizeCopy)) */
             if ((xCopy[iCopy-1]) && (!xCopy[iCopy-sizeCopy])) {
               BlobLabels[i] = BlobLabels[i-1];
             } else {   /* ((xCopy[iCopy-1]) && (!xCopy[iCopy-sizeCopy])) */
              /*
               * if you are here, then both causal neighbors are already
               * labeled
               */
               BlobLabels[i] = BlobLabels[i-1];
               if (BlobLabels[i-1] != BlobLabels[i-size]) {
                 /* Causal neighbors have different labels */
                 /* Combine the two blobs into a single new blob */
                 /* Keep the label that is smallest numerically */
                 /* Kill the other label.  Shift all labels greater than  */
                 /*   the killed label so that the used labels remain     */
                 /*   contiguous.                                         */
                 if (BlobLabels[i-1] < BlobLabels[i-size]) {
```

```
                  KillLabel = BlobLabels[i-size];
                  KeepLabel = BlobLabels[i-1];
                } else {
                  KillLabel = BlobLabels[i-1];
                  KeepLabel = BlobLabels[i-size];
                }
                for (j=0; j <= i; j++) {
                  if (BlobLabels[j] == KillLabel) {
                    BlobLabels[j] = KeepLabel;
                  } else {                 /* if BlobLabels[j] */
                    if (BlobLabels[j] > KillLabel) {
                      BlobLabels[j]--;
                    }
                  }
                }    /* for j */
                NextLabel--;
              }      /* if BlobLabels[i-1] != BlobLabels[i-size]) */
            } /* else of ((xCopy[iCopy-1]) && (!xCopy[iCopy-sizeCopy])) */
          }   /* else of ((!xCopy[iCopy-1]) && (xCopy[iCopy-sizeCopy])) */
        }     /* else of ((!xCopy[iCopy-1]) && (!xCopy[iCopy-sizeCopy])) */
      }       /* if xCopy[iCopy] */
    }         /* for col */
  }           /* for row */

  free(xCopy);         /* garbage collection */
  return(--NextLabel);
} /*----------------  BlobColor  ----------------------------------------*/


/*----------------------------------------------------------------------
 * BlobCount
 *
 *   Function performs blob counting.  Int array BlobLabels contains the blob
 *   colors (labels).  Int array BlobCounts is returned with the counts.
 *   MaxLabel is the dimension of array BlobCounts and also the largest blob
 *   color (label) that was assigned in the BlobColor routine.
 *
 *  jph 12 April 2002
 *
 ----------------------------------------------------------------------*/

void BlobCount(size,BlobLabels,MaxLabel,BlobCounts)

  int      size;          /* row/col dimension of image x */
  int     *BlobLabels;    /* computed array of blob labels (colors) */
  int      MaxLabel;      /* one less than dimension of BlobLabels array */
  int     *BlobCounts;    /* array of blob counts to be returned */

{

  int      N;             /* number of pixels = size * size */
  int      i;             /* counter */

  N = size * size;

 /*
  * Initialize BlobCounts array
  */
  for (i=0; i <= MaxLabel; i++) {
    BlobCounts[i] = 0;
  }

 /*
  * Count the blob labels (but don't count the background -- the blob with
  *     label zero...)
  */
  for (i=0; i < N; i++) {
    if (BlobLabels[i]) {
```

```
      BlobCounts[BlobLabels[i]]++;
    }
  }
  return;
} /*----------------  BlobCount  ----------------------------------------*/


/*----------------------------------------------------------------------
 * RemoveMinorRegions
 *
 *   Function removes minor regions in a blob colored image.  Logic one pixels
 *   in the image that have been labeled as part of the largest blob are
 *   retained.  Logic one pixels that do not belong to the largest blob
 *   are set to zero.  Blob counts and labels for the minor regions are also
 *   reset to zero.
 *
 *  jph 15 April 2002
 *
 -----------------------------------------------------------------------*/

void RemoveMinorRegions(x,size,BlobLabels,MaxLabel,BlobCounts)

  BYTE    *x;             /* input image */
  int      size;          /* row/col dimension of image x */
  int     *BlobLabels;    /* computed array of blob labels (colors) */
  int      MaxLabel;      /* one less than dimension of BlobLabels array */
  int     *BlobCounts;    /* array of blob counts to be returned */

{

  int      N;             /* number of pixels = size * size */
  int      i;             /* counter */
  int      MaxCountLabel; /* label of blob containing the most pixels */
  int      MaxCount;      /* number of pixels in the largest blob */

  N = size * size;

 /*
  * Find out which blob has the largest count
  */
  for (MaxCountLabel=MaxCount=i=0; i <= MaxLabel; i++) {
    if (BlobCounts[i] > MaxCount) {
      MaxCount = BlobCounts[i];
      MaxCountLabel = i;
    }
  }

 /*
  * Zero out the counts of the minor blobs
  */
  for (i=0; i <= MaxLabel; i++) {
    if (i != MaxCountLabel) {
      BlobCounts[i] = 0;
    }
  }

 /*
  * Zero out the pixels and labels of the minor blobs
  */
  for (i=0; i < N; i++) {
    if (BlobLabels[i] != MaxCountLabel) {
      x[i] = LOGIC_ZERO;
      BlobLabels[i] = 0;
    }
  }

  return;
} /*----------------  RemoveMinorRegions  -------------------------------*/
```

```
/*----------------------------------------------------------------------
 * WriteAsciiArrayBYTE
 *
 *   temporary diagnostic routine writes a square 2D BYTE array to the
 *   console in ASCII.
 *
 *  jph 12 April 2002
 *
 -----------------------------------------------------------------------*/

void WriteAsciiArrayBYTE(x,size)

  BYTE    *x;                    /* input array */
  int      size;                 /* num rows/cols in image */
{

  int      row;                  /* row counter */
  int      col;                  /* column counter */

 /*
  * Loop over rows and cols, write out ascii data
  */
  for (row=0; row < size; row++) {
    printf("%4d",x[row*size + 0]);  /* 1st entry; no leading blank */
    for (col=1; col < size; col++) {
      printf("%5d",x[row*size + col]); /* rest w/ leading blank */
    }
    printf("\n");        /* eol */
  }
  return;
} /*---------------------  WriteAsciiArrayBYTE  ----------------------------*/

/*----------------------------------------------------------------------
 * WriteAsciiArrayInt
 *
 *   temporary diagnostic routine writes a square 2D int array to the
 *   console in ASCII.
 *
 *  jph 12 April 2002
 *
 -----------------------------------------------------------------------*/

void WriteAsciiArrayInt(x,size)

  int     *x;                    /* input array */
  int      size;                 /* num rows/cols in image */
{

  int      row;                  /* row counter */
  int      col;                  /* column counter */

 /*
  * Loop over rows and cols, write out ascii data
  */
  for (row=0; row < size; row++) {
    printf("%9d",x[row*size + 0]);  /* 1st entry; no leading blank */
    for (col=1; col < size; col++) {
      printf("%10d",x[row*size + col]); /* rest w/ leading blank */
    }
    printf("\n");        /* eol */
  }
  return;
} /*---------------------  WriteAsciiArrayInt  ---------------------------*/

/*----------------------------------------------------------------------
 * InvertBYTEImage
 *
```

```
 *   Invert a BYTE binary image.
 *
 *  jph 12 April 2002
 *
 -------------------------------------------------------------------------*/

void InvertBYTEImage(x,N)

  BYTE    *x;                    /* input array */
  int      N;                    /* num pixels in image */
{

  int      i;                    /* counter */

  for (i=0; i < N; i++) {
    if (x[i]) {
      x[i] = LOGIC_ZERO;
    } else {
      x[i] = LOGIC_ONE;
    }
  }
  return;
} /*---------------------  InvertBYTEImage  ------------------------------*/

/*----------------------------------------------------------------------
 * i2bdbx
 *
 *   input an int image of blob labels.  Convert to a BYTE image and perform
 *   full-scale contrast stretch for display.  Return result as BYTE image.
 *
 *  jph 12 April 2002
 *
 -------------------------------------------------------------------------*/

BYTE *i2bdbx(x,N,size)

  int     *x;                    /* input int array of Blob Labels */
  int      N;                    /* num pixels in image */
  int      size;                 /* row/col dimension of image */
{

  BYTE    *y;                    /* output byte image */
  int      i;                    /* counter */
  int      row;                  /* row counter */
  int      col;                  /* col counter */
  int      min,max;              /* min and max pixel values in x */
  float    factor;               /* multiplicative factor */

  if ((y = (BYTE *)malloc(N*sizeof(BYTE))) == NULL) {
    printf("\ni2bdbx: free store exhausted.\n");
    exit(-1);
  }

  max = min = x[0];
  for (i=0; i < N; i++) {
    if (x[i] > max) max = x[i];
    else if (x[i] < min) min = x[i];
  }

  factor = (float) 255.0 / (float)(max - min);

 /*
  * Since there will generally be more than 256 blobs, we will invert the
  *   blob image and put approximate edges between the labeled regions to
  *   make it easier to look at the "blob" image.
  */
  for (i=row=0; row < size; row++) {
    for (col=0; col < size; col++,i++) {
```

```
      y[i] = 255 - (BYTE)((float)(x[i] - min) * factor);
      if (col && row) {
        if ((x[i] != x[i-1]) ||  (x[i] != x[i-size])) {
          /* this pixel is on the boundary btwn 2 blobs: make it black */
          y[i] = 0;
        }
      }
    }    /* for col */
  }      /* for row */

 /*
  * Also, let's add a border to the image; since we inverted it it may be
  *    mostly white -- and a border will make it look better when we print.
  */
  for (i=0; i < size; i++) {
    y[i] = y[i*size] = y[i*size+size-1] = y[N-i-1] = 0;
  }
  return(y);
} /*---------------------  i2bdbx  --------------------------------------*/

/*----------------------------------------------------------------------
 * FuseForDisplay
 *
 *   Routine combines final blob coloring result with original grey scale
 *   image to make a nice looking display image that conveys the blob
 *   coloring result.
 *
 *   The output image is set equal to 255 everywhere.   For LOGICAL_ONE
 *   pixels in the blob coloring final result, the output image is set equal
 *   to the original image.  Finally, a black border is put onto the
 *   output image.
 *
 *  jph 12 April 2002
 *
 ----------------------------------------------------------------------*/

BYTE *FuseForDisplay(x,OrigFn,size,N,DisplayFuseFn)

  BYTE    *x;                    /* final blob coloring result: binary */
  char    *OrigFn;               /* file name of original grey scale image */
  int      size;                 /* row/col dimension of image */
  int      N;                    /* num pixels in image */
  char    *DisplayFuseFn;        /* output filename for the fused result */
{

  BYTE    *y;                    /* output byte image */
  BYTE    *Orig;                 /* original grey scale image */
  int      i;                    /* counter */

 /*
  * Allocate; read the original image
  */
  if ((y = (BYTE *)malloc(N*sizeof(BYTE))) == NULL) {
    printf("\nDisplayFuseFn: free store exhausted.\n");
    exit(-1);
  }
  if ((Orig = (BYTE *)malloc(N*sizeof(BYTE))) == NULL) {
    printf("\nDisplayFuseFn: free store exhausted.\n");
    exit(-1);
  }
  disk2byte(Orig,size,size,OrigFn);

 /*
  * Set y=orignal if blob result is ONE, otherwise y=ONE.
  */
  for (i=0; i < N; i++) {
    if (x[i]) {
      y[i] = Orig[i];
```

```
    } else {
      y[i] = LOGIC_ONE;
    }
  }

 /*
  * Put a nice border on y
  */
  for (i=0; i < size; i++) {
    y[i] = y[i*size] = y[i*size+size-1] = y[N-i-1] = 0;
  }

 /*
  * Collect Garbage and return
  */
  free(Orig);
  return(y);
} /*--------------------  FuseForDisplay  ------------------------------*/

/*----------------------------------------------------------------------
 * disk2byte.c
 *
 *   function reads an unsigned char (byte) image from disk
 *
 *
 *  jph 15 June 1992
 *    12 April 2002: revision for improved error handling, jph
 *
 ------------------------------------------------------------------------*/

void disk2byte(x,row_dim,col_dim,fn)

  BYTE    *x;             /* image to be read */
  int     row_dim;        /* row dimension of x */
  int     col_dim;        /* col dimension of x */
  char   *fn;             /* filename */
{

  int fd;               /* file descriptor */
  int n_bytes;          /* number of bytes to read */

 /*
  * detect zero dimension input
  */
  if ((row_dim==0) || (col_dim==0)) {
    printf("\ndisk2byte.c : row_dim=%d, col_dim=%d !\n\n",row_dim,col_dim);
    exit(-1);
  }

 /*
  * create and open the file
  */
  if ((fd = open(fn, O_RDONLY))==-1) {
    printf("\ndisk2byte.c : could not open %s !\n\n",fn);
    exit(-1);
  }

 /*
  * read image data from the file
  */
  n_bytes = row_dim * col_dim * sizeof(unsigned char);
  if (read(fd,x,n_bytes) != n_bytes) {
    printf("\ndisk2byte.c : complete read of %s did not succeed.\n\n",fn);
    exit(-1);
  }

 /*
  * close file and return
```

```
  */
  if (close(fd) == -1) printf("\ndisk2byte.c : error closing %s.",fn);
  return;
}



/*---------------------------------------------------------------------
 * byte2disk.c
 *
 *   function writes an unsigned char (byte) image to disk
 *
 *
 *  jph 15 June 1992
 *
 ----------------------------------------------------------------------*/

void byte2disk(x,row_dim,col_dim,fn)

  BYTE  *x;              /* image to be written */
  int   row_dim;         /* row dimension of x */
  int   col_dim;         /* col dimension of x */
  char *fn;              /* filename */
{

  int fd;               /* file descriptor */
  int n_bytes;          /* number of bytes to read */

 /*
  * detect zero dimension input
  */
  if ((row_dim==0) || (col_dim==0)) return;

 /*
  * create and open the file
  */
  if ((fd = open(fn, O_WRONLY | O_CREAT | O_TRUNC, 0644))==-1) {
    printf("\nbyte2disk.c : could not open %s !",fn);
    return;
  }

 /*
  * write image data to the file
  */
  n_bytes = row_dim * col_dim * sizeof(unsigned char);
  if (write(fd,x,n_bytes) != n_bytes) {
    printf("\nbyte2disk.c : complete write of %s did not succeed.",fn);
  }

 /*
  * close file and return
  */
  if (close(fd) == -1) printf("\nbyte2disk.c : error closing %s.",fn);
  return;
}

/*
 * bthres2dir:
 *
 *   Turn a byte image into a binary image by thresholding at a specified
 *   value.  The parameter direction determines if values above threshold
 *   are output as LOGIC ONE or if values below threshold are output as
 *   LOGIC ONE.  The output uses one byte per pixel.  LOGIC ONE is represented
 *   by the value 0xFF and LOGIC ZERO is represented by the value 0x00.
 *
 *   If direction = 1, then:
 *       For pixels that exceed threshold, the output is LOGIC ONE.  Otherwise
 *       the output is LOGIC ZERO.
```

```
 *
 *   If direction = 0, then:
 *       For pixels that are below threshold, the output is LOGIC ONE.
 *       Otherwise the output is LOGIC ZERO.
 *
 *   The input image must be square.
 *
 *
 *
 * 4/16/2002 jph
 *
 */

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>

#define BYTE unsigned char
#define LOGIC_ONE (0xff)
#define LOGIC_ZERO (0x00)

/*
 * Function Prototypes (forward declarations)
 */
void disk2byte();
void byte2disk();


main(argc,argv)

  int     argc;
  char   *argv[];
{

  int      i;                    /* counter */
  int      size;                 /* num rows/cols in image */
  int      direction;            /* 1/0 = above/below pixels get LOGIC ONE */
  BYTE     thresh;               /* threshold value to be used */
  BYTE     above_threshold;      /* outpt val for pixel exceeding threshold */
  BYTE     below_threshold;      /* outpt val for pixel below threshold */
  BYTE    *x;                    /* input image */
  BYTE    *y;                    /* output image */
  char    *infile;               /* input filename */
  char    *outfile;              /* output filename */

 /*
  * Check for proper invocation, parse args
  */
  if (argc != 6) {
    printf(
   "\n%s: converts sizeXsize byte image to a binary byte image by thesholding.",
       argv[0]);
    printf("\n      direction=1: pixels above threshold are output as 0xFF");
    printf("\n      direction=0: pixels below threshold are output as 0xFF");
    printf("\nUsage: %s size thresh direction infile outfile\n",argv[0]);
    exit(0);
  }
  size = atoi(argv[1]);
  thresh = (BYTE)atoi(argv[2]);
  direction = atoi(argv[3]);
  infile = argv[4];
  outfile = argv[5];

 /*
  * Allocate image arrays
  */
```

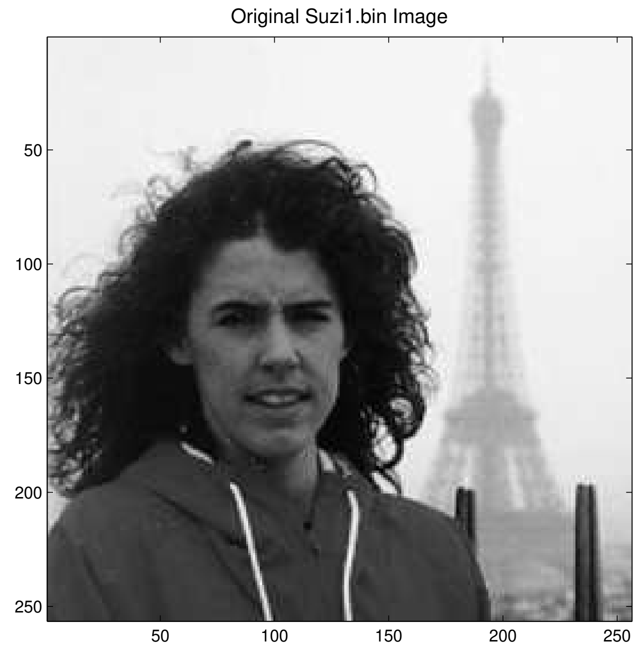
```
  if ((x = (BYTE *)malloc(size*size*sizeof(BYTE))) == NULL) {
    printf("\n%s: free store exhausted.\n",argv[0]);
    exit(-1);
  }
  if ((y = (BYTE *)malloc(size*size*sizeof(BYTE))) == NULL) {
    printf("\n%s: free store exhausted.\n",argv[0]);
    exit(-1);
  }

 /*
  * Read the input image
  */
  disk2byte(x,size,size,infile);

 /*
  * Threshold the image
  */
  if (direction) {
    above_threshold = LOGIC_ONE;
    below_threshold = LOGIC_ZERO;
  } else {
    above_threshold = LOGIC_ZERO;
    below_threshold = LOGIC_ONE;
  }
  for (i=0; i<size*size; i++) {
    if (x[i] < thresh) {
      y[i] = below_threshold;
    }
    else {
      y[i] = above_threshold;
    }
  }

 /*
  * Write the output image
  */
  byte2disk(y,size,size,outfile);

  return;
} /*----------------  Main  ----------------------------------------------*/

/*----------------------------------------------------------------------
 * disk2byte.c
 *
 *   function reads an unsigned char (byte) image from disk
 *
 *
 *  jph 15 June 1992
 *    12 April 2002: revision for improved error handling, jph
 *
 -----------------------------------------------------------------------*/

void disk2byte(x,row_dim,col_dim,fn)

  BYTE    *x;             /* image to be read */
  int     row_dim;        /* row dimension of x */
  int     col_dim;        /* col dimension of x */
  char   *fn;             /* filename */
{

  int fd;               /* file descriptor */
  int n_bytes;          /* number of bytes to read */

 /*
  * detect zero dimension input
  */
  if ((row_dim==0) || (col_dim==0)) {
    printf("\ndisk2byte.c : row_dim=%d, col_dim=%d !\n\n",row_dim,col_dim);
```

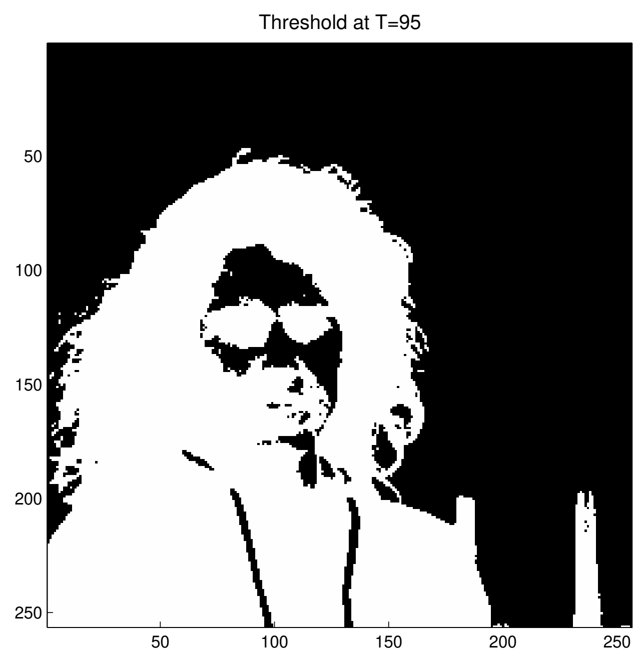
```
    exit(-1);
  }

 /*
  * create and open the file
  */
  if ((fd = open(fn, O_RDONLY))==-1) {
    printf("\ndisk2byte.c : could not open %s !\n\n",fn);
    exit(-1);
  }

 /*
  * read image data from the file
  */
  n_bytes = row_dim * col_dim * sizeof(unsigned char);
  if (read(fd,x,n_bytes) != n_bytes) {
    printf("\ndisk2byte.c : complete read of %s did not succeed.\n\n",fn);
    exit(-1);
  }

 /*
  * close file and return
  */
  if (close(fd) == -1) printf("\ndisk2byte.c : error closing %s.",fn);
  return;
}

/*----------------------------------------------------------------------
 * byte2disk.c
 *
 *   function writes an unsigned char (byte) image to disk
 *
 *
 *  jph 15 June 1992
 *
 ------------------------------------------------------------------------*/

void byte2disk(x,row_dim,col_dim,fn)

  BYTE  *x;              /* image to be written */
  int   row_dim;         /* row dimension of x */
  int   col_dim;         /* col dimension of x */
  char *fn;              /* filename */
{

  int fd;               /* file descriptor */
  int n_bytes;          /* number of bytes to read */

 /*
  * detect zero dimension input
  */
  if ((row_dim==0) || (col_dim==0)) return;

 /*
  * create and open the file
  */
  if ((fd = open(fn, O_WRONLY | O_CREAT | O_TRUNC, 0644))==-1) {
    printf("\nbyte2disk.c : could not open %s !",fn);
    return;
  }

 /*
  * write image data to the file
  */
  n_bytes = row_dim * col_dim * sizeof(unsigned char);
  if (write(fd,x,n_bytes) != n_bytes) {
    printf("\nbyte2disk.c : complete write of %s did not succeed.",fn);
  }
```

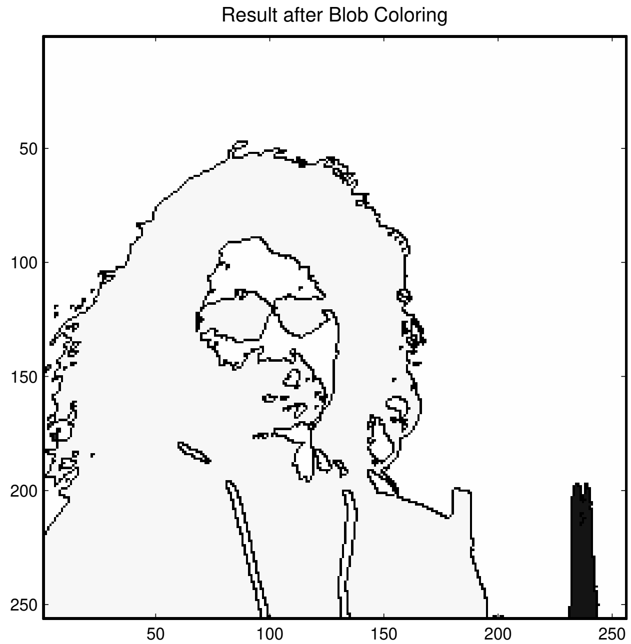
```
 /*
  * close file and return
  */
  if (close(fd) == -1) printf("\nbyte2disk.c : error closing %s.",fn);
  return;
}
```

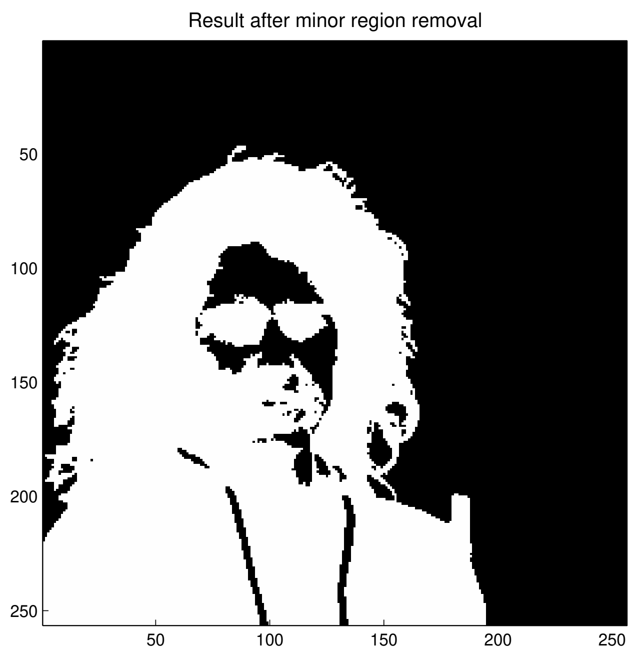
Matlab Solution:

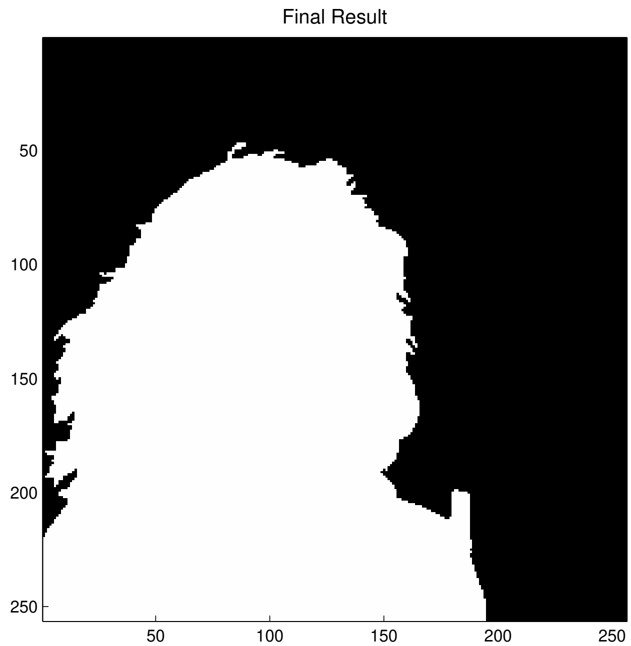
Threshold selection is explained in the **C Solution**.

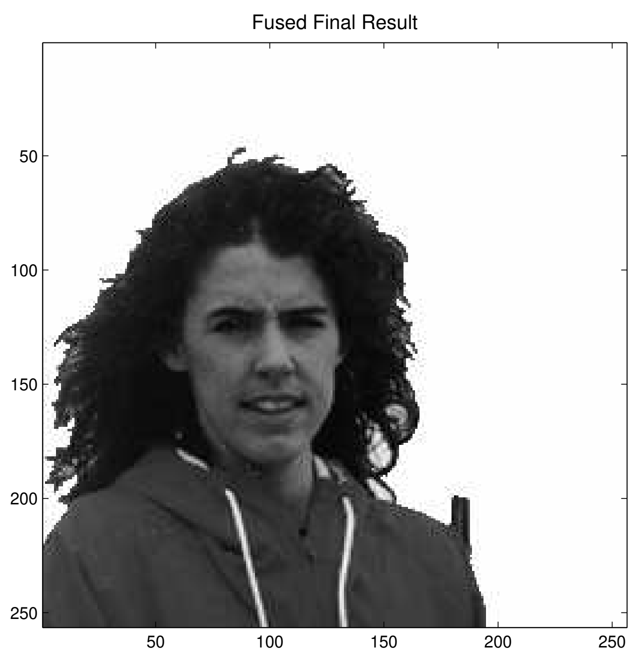


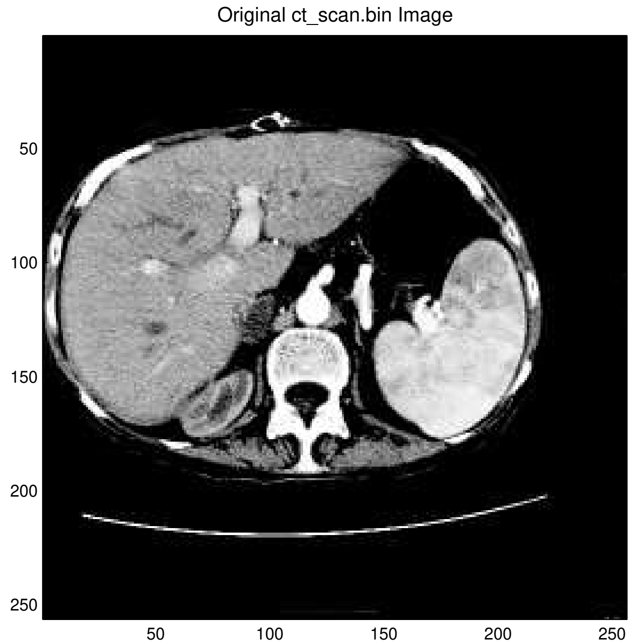


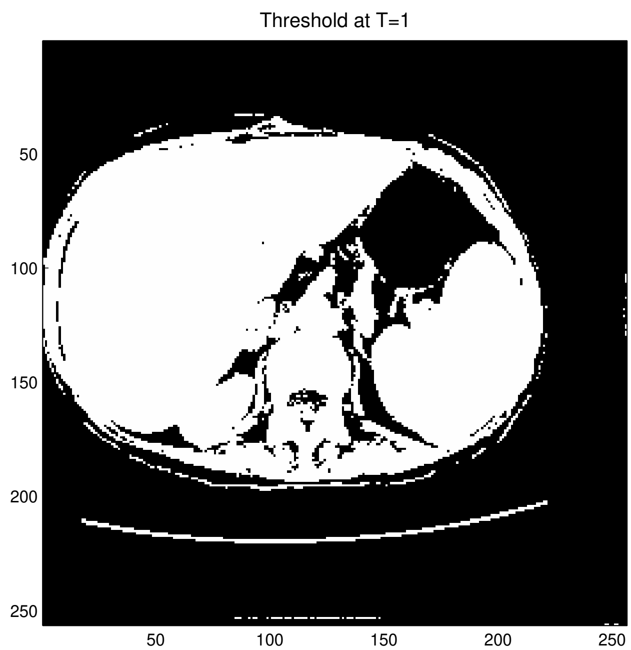

Result after Blob Coloring
50
100
150
200
250
50
100
150
200
250

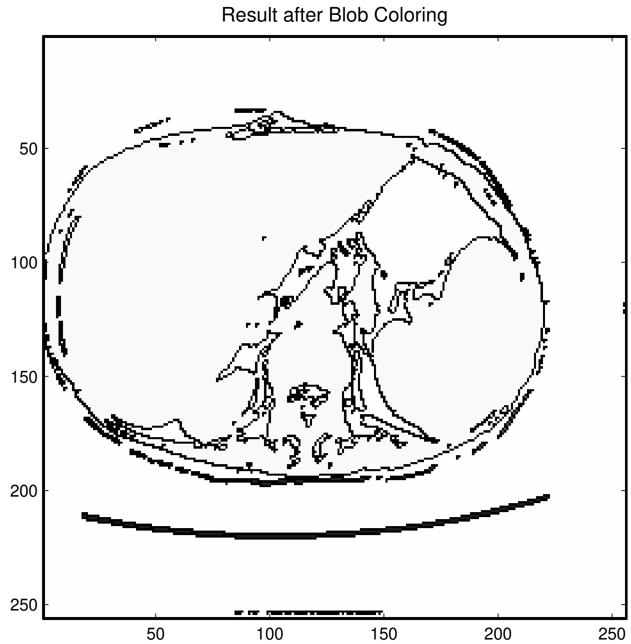

Result after minor region removal
50
100
150
200
250
50
100
150
200
250

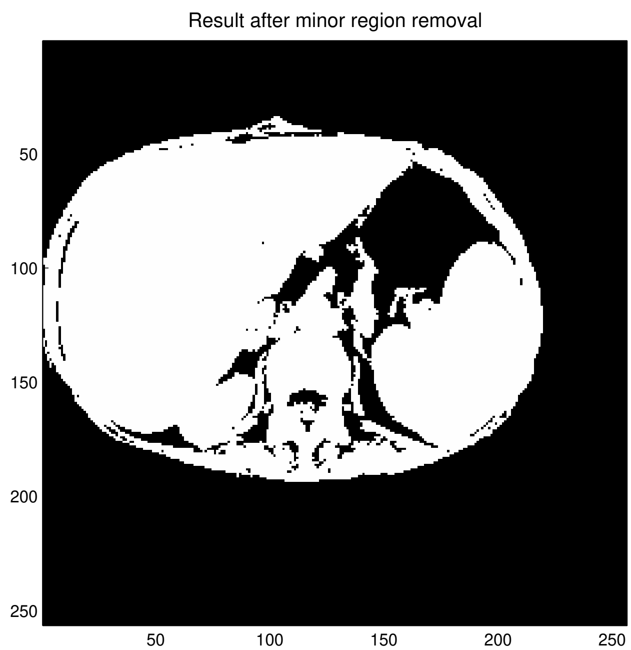

Final Result
50
100
150
200
250
50
100
150
200
250

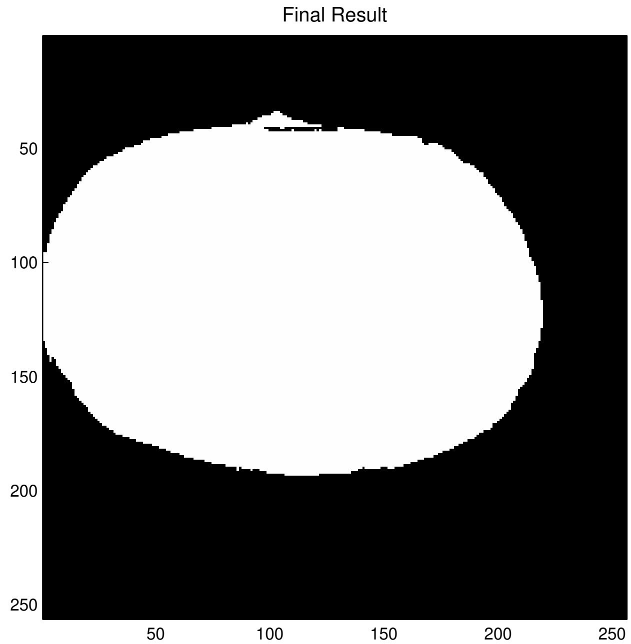

Fused Final Result
50
100
150
200
250
50
100
150
200
250

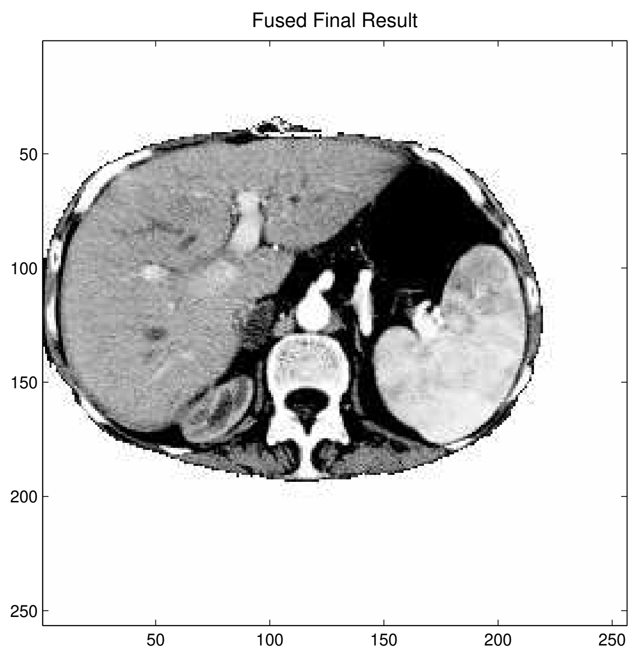

Original ct_scan.bin Image
50
100
150
200
250
50
100
150
200
250


Threshold at T=1
50
100
150
200
250
50
100
150
200
250


Result after Blob Coloring
50
100
150
200
250
50
100
150
200
250


Result after minor region removal
50
100
150
200
250
50
100
150
200
250


Final Result
50
100
150
200
250
50
100
150
200
250


Fused Final Result
50
100
150
200
250
50
100
150
200
250

Matlab m-file listing:

```
%
% hw02.m
%
% 04/20/02 jph
%
% 02/05/11: for semesters up until SP 2011 this was hw03.  Starting in SP 2011,
%           it is now hw02.  jph
% 01/20/14: took out some loops to make it run faster. jph
%

size = 256;                        % use this line for running on a big image
%size = 8;                         % use this line for DEBUG on small test image
BlobLabels = zeros(size,size);

%
% Suzi1
%

%
% Read Suzi1.bin, transpose, display, and print
%

% The following lines are for running on the big image
fidSuzi1=fopen('Suzi1.bin','r');
[Suzi1,junk] = fread(fidSuzi1,[size,size],'uchar');
Suzi1 = Suzi1';
% the following lines are for DEBUG -- instead of reading the big image,
%    make a small 8x8 test image for debugging.
%Suzi1 = zeros(size,size);
%Suzi1(1,1) = 128;
%Suzi1(1,2) = 128;
%Suzi1(2,1) = 128;
%Suzi1(2,2) = 128;
%
%Suzi1(2,5) = 128;
%Suzi1(3,5) = 128;
%
%Suzi1(3,7) = 128;
%Suzi1(3,8) = 128;
%
%Suzi1(5,1) = 128;
%Suzi1(6,1) = 128;
%Suzi1(7,1) = 128;
%Suzi1(7,2) = 128;
%Suzi1(7,3) = 128;
%Suzi1(7,4) = 128;
%Suzi1(7,5) = 128;
%Suzi1(6,5) = 128;
%Suzi1(5,5) = 128;
%
%Suzi1(5,3) = 128;
%
%Suzi1(5,8) = 128;
%
%Suzi1(6,7) = 128;
%
%Suzi1(8,7) = 128;
% END of the DEBUG section (which makes the small 8x8 test image)

figure(1);colormap(gray(256));
image(Suzi1);axis('image');
title('Original Suzi1.bin Image','FontSize',12);
print -deps MSuzi1.eps;
pause;

%
```

```
% Threshold at the empirically determined value thresh=95.
%   Thresholded image = 255 if original < thresh.
%
x = zeros(size,size);

thresh = 95;

x(Suzi1 < thresh) = 255;             % this line for running on big image
%x(Suzi1 > thresh) = 255;            % this line for DEBUG on 8x8 test image

%
% Display and print thresholding result
%
figure(2);colormap(gray(256));
image(x);axis('image');
title('Threshold at T=95','FontSize',12);
print -deps MSuzi1Thresh.eps;
pause;

%
% Perform Blob Coloring on the thresholding result; display and print
%
[BlobLabels,MaxLabel] = BlobColor(x,size,BlobLabels);
y = i2bdbx(BlobLabels,size);
figure(3);colormap(gray(256));
image(y);axis('image');
title('Result after Blob Coloring','FontSize',12);
print -deps MSuzi1Blob.eps;
pause;

%
% Perform Blob Counting, remove minor regions, display, and print
%
BlobCounts = BlobCount(size,BlobLabels,MaxLabel);
x = RemoveMinorRegions(x,size,BlobLabels,BlobCounts);
figure(4);colormap(gray(256));
image(x);axis('image');
title('Result after minor region removal','FontSize',12);
print -deps MSuzi1BlobMinor.eps;
pause;

%
% Invert image and repeat blob coloring and minor region removal
%
x = 255 - x;
[BlobLabels,MaxLabel] = BlobColor(x,size,BlobLabels);
BlobCounts = BlobCount(size,BlobLabels,MaxLabel);
x = RemoveMinorRegions(x,size,BlobLabels,BlobCounts);

%
% Invert to get the final result
%
x = 255 - x;
figure(5);colormap(gray(256));
image(x);axis('image');
title('Final Result','FontSize',12);
print -deps MSuzi1Final.eps;
pause;

%
% Make a special fused image that combines the original with the final blob
%   coloring result... this is just to make a fancy display.
%
y = 255 - ((x .* (255 - Suzi1)) / 255);
figure(6);colormap(gray(256));
image(y);axis('image');
title('Fused Final Result','FontSize',12);
print -deps MSuzi1Fuse.eps;
```

```
pause;




%
% ct_scan
%

size = 256;
BlobLabels = zeros(size,size);

%
% Read ct_scan.bin, transpose, display, and print
%

fidct_scan=fopen('ct_scan.bin','r');
[ct_scan,junk] = fread(fidct_scan,[size,size],'uchar');
ct_scan = ct_scan';
figure(11);colormap(gray(256));
image(ct_scan);axis('image');
title('Original ct\_scan.bin Image','FontSize',12);
print -deps Mct_scan.eps;
pause;

%
% Threshold at the empirically determined value thresh=1.
%   Thresholded image = 255 if original >= thresh.
%
x = zeros(size,size);
thresh = 1;
x(ct_scan >= thresh) = 255;

%
% Display and print thresholding result
%
figure(12);colormap(gray(256));
image(x);axis('image');
title('Threshold at T=1','FontSize',12);
print -deps Mct_scanThresh.eps;
pause;

%
% Perform Blob Coloring on the thresholding result; display and print
%
[BlobLabels,MaxLabel] = BlobColor(x,size,BlobLabels);
y = i2bdbx(BlobLabels,size);
figure(13);colormap(gray(256));
image(y);axis('image');
title('Result after Blob Coloring','FontSize',12);
print -deps Mct_scanBlob.eps;
pause;

%
% Perform Blob Counting, remove minor regions, display, and print
%
BlobCounts = BlobCount(size,BlobLabels,MaxLabel);
x = RemoveMinorRegions(x,size,BlobLabels,BlobCounts);
figure(14);colormap(gray(256));
image(x);axis('image');
title('Result after minor region removal','FontSize',12);
print -deps Mct_scanBlobMinor.eps;
pause;

%
% Invert image and repeat blob coloring and minor region removal
%
```

```
x = 255 - x;
[BlobLabels,MaxLabel] = BlobColor(x,size,BlobLabels);
BlobCounts = BlobCount(size,BlobLabels,MaxLabel);
x = RemoveMinorRegions(x,size,BlobLabels,BlobCounts);

%
% Invert to get the final result
%
x = 255 - x;
figure(15);colormap(gray(256));
image(x);axis('image');
title('Final Result','FontSize',12);
print -deps Mct_scanFinal.eps;
pause;

%
% Make a special fused image that combines the original with the final blob
%   coloring result... this is just to make a fancy display.
%
y = 255 - ((x .* (255 - ct_scan)) / 255);
figure(16);colormap(gray(256));
image(y);axis('image');
title('Fused Final Result','FontSize',12);
print -deps Mct_scanFuse.eps;
pause;

%
% BlobColor.m
%
%   Function performs connected components labeling (blob coloring) on
%   a square binary image.  The largest label assigned by the algorithm
%   is returned.
%
% jph 20 April 2002
%   01/20/14: took out some loops to make it run faster.  jph
%

function [BlobLabels,MaxLabel] = BlobColor(x,size,BlobLabels)

sizeCopy = size + 1;
NextLabel = 1;

%
% init the BlobLabels array
%
BlobLabels(:) = 0;

%
% Make xCopy - a copy of x with one extra row and col prepended
%
xCopy = zeros(sizeCopy,sizeCopy);
xCopy(2:sizeCopy,2:sizeCopy) = x;

%
%  Loop on rows and columns, color blobs
%   -the xCopy array has a copy of the original binary image x, but with
%    an extra first row of all zeros and an extra first col of all zeros.
%   -we can now loop over rows and cols starting at TWO instead of ONE in the
%    xCopy array and we don't have to worry about addressing off the left
%    side or the top of the array in doing the algorithm on page 2.44 of the
%    course notes (page 2.31 of the old notes).
%   -doing this requires us to keep track of two sets of row and col counters.
%    One set loops over rows/cols in xCopy starting at TWO.  The other set
%    simultaneoulsy loops over the corresponding points in the BlobLabels
%    array starting each row/col loop at ONE.
%
rowCopy = 2;
for row=1:size
```

```
colCopy = 2;
for col=1:size
  if (xCopy(rowCopy,colCopy) == 255)
    if ((xCopy(rowCopy,colCopy-1) == 0) & (xCopy(rowCopy-1,colCopy) == 0))
      % This is case 1 on page 2.44 of the notes:
      %   Neither the upper nor the left neighbor already have labels
      BlobLabels(row,col) = NextLabel;
      NextLabel = NextLabel + 1;
    else                    % else of if case 1
      if ((xCopy(rowCopy,colCopy-1)==0) & (xCopy(rowCopy-1,colCopy)==255))
        % upper neighbor ONLY already labeled; call this case 2
        BlobLabels(row,col) = BlobLabels(row-1,col);
      else                  % else of if case 2
        if ((xCopy(rowCopy,colCopy-1)==255) & (xCopy(rowCopy-1,colCopy)==0))
          % left neighbor ONLY already labeled; call this case 3
          BlobLabels(row,col) = BlobLabels(row,col-1);
        else                % else of if case 3
          %
          % if you are here, then both causal neighbors are already labeled
          %
          BlobLabels(row,col) = BlobLabels(row,col-1);
          if (BlobLabels(row,col-1) ~= BlobLabels(row-1,col))
            %
            % Causal neighbors have different labels
            % Combine the two blobs into a single new blob
            % Keep the label that is smallest numerically
            % Kill the other label.  Shift all labels greater than
            %   the killed label so that the used labels remain
            %   contiguous.
            %
            if (BlobLabels(row,col-1) < BlobLabels(row-1,col))
              KillLabel = BlobLabels(row-1,col);
              KeepLabel = BlobLabels(row,col-1);
            else
              KillLabel = BlobLabels(row,col-1);
              KeepLabel = BlobLabels(row-1,col);
            end
            %
            % Shift labels on all rows before the current row
            %
            for rrow=1:row-1
              for ccol=1:size
                if (BlobLabels(rrow,ccol) == KillLabel)
                  BlobLabels(rrow,ccol) = KeepLabel;
                else        % else of if BlobLabels == KillLabel
                  if (BlobLabels(rrow,ccol) > KillLabel)
                    BlobLabels(rrow,ccol) = BlobLabels(rrow,ccol) - 1;
                  end       % if BlobLabels > KillLabel
                end         % end else of if BlobLabels == KillLabel
              end           % for ccol
            end             % for rrow
            %
            % Shift labels on the current row
            %
            rrow = row;
            for ccol=1:col
              if (BlobLabels(rrow,ccol) == KillLabel)
                BlobLabels(rrow,ccol) = KeepLabel;
              else        % else of if BlobLabels == KillLabel
                if (BlobLabels(rrow,ccol) > KillLabel)
                  BlobLabels(rrow,ccol) = BlobLabels(rrow,ccol) - 1;
                end       % if BlobLabels > KillLabel
              end         % end else of if BlobLabels == KillLabel
            end           % for ccol
            NextLabel = NextLabel - 1;
          end               % if BlobLabels(row,col-1)~=BlobLabels(row-1,col)
        end                 % end else of if case 3
      end                   % end else of if case 2
```

```
      end                      % end else of if case 1
    end                        % if (xCopy(rowCopy,colCopy) == 255)
    colCopy = colCopy + 1;
  end                          % for col
  rowCopy = rowCopy + 1;
end                            % for row

MaxLabel = NextLabel-1;

return;

%
% BlobCount.m
%
% Function performs blob counting.  Array BlobLabels contains the blob
% colors (labels).  Array BlobCounts is returned with the counts.
% MaxLabel is the dimension of array BlobCounts and also the largest blob
% color (label) that was assigned in the BlobColor routine.
%
% jph 20 April 2002
%

function [BlobCounts] = BlobCount(size,BlobLabels,MaxLabel)

%
% Count the blob labels; but don't count the background -- the blob with label
%   zero.
%
BlobCounts = zeros(MaxLabel,1);
for row=1:size
  for col=1:size
    if (BlobLabels(row,col) > 0)
      BlobCounts(BlobLabels(row,col)) = BlobCounts(BlobLabels(row,col)) + 1;
    end
  end
end

return;

%
% i2bdbx.m
%
% input image of blob labels.  Process for display:
%    - Full Scale Contrast Stretch
%    - Invert scaled Blob Colors
%    - add edges between the blobs
%
% jph 20 April 2002
%    01/20/14: took out some loops to make it run faster. jph
%

function [y] = i2bdbx(x,size)

%
% allocate output image y
%
y = zeros(size,size);

%
% Find min, max, and scale factor and do full-scale stretch.
%
xMax = max(max(x));
xMin = min(min(x));
ScaleFactor = 255.0 / (xMax - xMin);
y = 255 - floor( (x - xMin) * ScaleFactor);

%
% Since there will generally be more than 256 blobs, we will invert the
```

```
% blob image and put approximate edges between the labeled regions to
% make it easier to look at the "blob" image.
%
for row=1:size
  for col=1:size
    if ((row > 1) & (col > 1))
      if ((x(row,col) ~= x(row,col-1)) | (x(row,col) ~= x(row-1,col)))
        % this pixel is on the boundary btwn 2 blobs: make it black
        y(row,col) = 0;
      end
    end
  end
end

%
% Also, let's add a border to the image; since we inverted it it may be
%    mostly white -- and a border will make it look better when we print.
%
y(1,1:size) = 0;
y(size,1:size) = 0;
y(1:size,1) = 0;
y(1:size,size) = 0;

return;


%
% RemoveMinorRegions.m
%
% Function removes minor regions in a blob colored image.  Logic one pixels
% in the image that have been labeled as part of the largest blob are
% retained.  Logic one pixels that do not belong to the largest blob
% are set to zero.
%
% jph 20 April 2002
%   01/20/14: took out loops to make it run faster.  jph
%

function [x] = RemoveMinorRegions(x,size,BlobLabels,BlobCounts)

[junk,MaxCountLabel] = max(BlobCounts);

x(BlobLabels ~= MaxCountLabel) = 0;

return;
```