```
/* source: http://marathon.csee.usf.edu/edge/edge_detection.html */
/* URL: ftp://figment.csee.usf.edu/pub/Edge_Comparison/source_code/canny.src */


/*******************************************************************************
* --------------------------------------------
*(c) 2001 University of South Florida, Tampa
* Use, or copying without permission prohibited.
* PERMISSION TO USE
* In transmitting this software, permission to use for research and
* educational purposes is hereby granted.  This software may be copied for
* archival and backup purposes only.  This software may not be transmitted
* to a third party without prior permission of the copyright holder. This
* permission may be granted only by Mike Heath or Prof. Sudeep Sarkar of
* University of South Florida (sarkar@csee.usf.edu). Acknowledgment as
* appropriate is respectfully requested.
*
*  Heath, M., Sarkar, S., Sanocki, T., and Bowyer, K. Comparison of edge
*    detectors: a methodology and initial study, Computer Vision and Image
*    Understanding 69 (1), 38-54, January 1998.
*  Heath, M., Sarkar, S., Sanocki, T. and Bowyer, K.W. A Robust Visual
*    Method for Assessing the Relative Performance of Edge Detection
*    Algorithms, IEEE Transactions on Pattern Analysis and Machine
*    Intelligence 19 (12),  1338-1359, December 1997.
*  ------------------------------------------------------
*
* PROGRAM: canny_edge
* PURPOSE: This program implements a "Canny" edge detector. The processing
* steps are as follows:
*
*   1) Convolve the image with a separable gaussian filter.
*   2) Take the dx and dy the first derivatives using [-1,0,1] and [1,0,-1]'.
*   3) Compute the magnitude: sqrt(dx*dx+dy*dy).
*   4) Perform non-maximal suppression.
*   5) Perform hysteresis.
*
* The user must input three parameters. These are as follows:
*
*   sigma = The standard deviation of the gaussian smoothing filter.
*   tlow  = Specifies the low value to use in hysteresis. This is a
*           fraction (0-1) of the computed high threshold edge strength value.
*   thigh = Specifies the high value to use in hysteresis. This fraction (0-1)
*           specifies the percentage point in a histogram of the gradient of
*           the magnitude. Magnitude values of zero are not counted in the
*           histogram.
*
* NAME: Mike Heath
*       Computer Vision Laboratory
*       University of South Floeida
*       heath@csee.usf.edu
*
* DATE: 2/15/96
*
* Modified: 5/17/96 - To write out a floating point RAW headerless file of
*                     the edge gradient "up the edge" where the angle is
*                     defined in radians counterclockwise from the x direction.
*                     (Mike Heath)
*******************************************************************************/
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <math.h>
#include <string.h>

#define VERBOSE 0

#define NOEDGE 255
#define POSSIBLE_EDGE 128
#define EDGE 0
#define BOOSTBLURFACTOR 90.0
#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

int read_pgm_image(char *infilename, unsigned char **image, int *rows,
    int *cols);
int write_pgm_image(char *outfilename, unsigned char *image, int rows,
    int cols, char *comment, int maxval);

void canny(unsigned char *image, int rows, int cols, float sigma,
         float tlow, float thigh, unsigned char **edge, char *fname);
void gaussian_smooth(unsigned char *image, int rows, int cols, float sigma,
        short int **smoothedim);
void make_gaussian_kernel(float sigma, float **kernel, int *windowsize);
void derrivative_x_y(short int *smoothedim, int rows, int cols,
        short int **delta_x, short int **delta_y);
void magnitude_x_y(short int *delta_x, short int *delta_y, int rows, int cols,
        short int **magnitude);
void apply_hysteresis(short int *mag, unsigned char *nms, int rows, int cols,
        float tlow, float thigh, unsigned char *edge);
void radian_direction(short int *delta_x, short int *delta_y, int rows,
    int cols, float **dir_radians, int xdirtag, int ydirtag);
double angle_radians(double x, double y);

main(int argc, char *argv[])
{
   char *infilename = NULL;  /* Name of the input image */
   char *dirfilename = NULL; /* Name of the output gradient direction image */
   char outfilename[128];    /* Name of the output "edge" image */
   char composedfname[128];  /* Name of the output "direction" image */
   unsigned char *image;     /* The input image */
   unsigned char *edge;      /* The output edge image */
   int rows, cols;           /* The dimensions of the image. */
   float sigma,              /* Standard deviation of the gaussian kernel. */
       tlow,               /* Fraction of the high threshold in hysteresis. */
       thigh;              /* High hysteresis threshold control. The actual
                          threshold is the (100 * thigh) percentage point
                          in the histogram of the magnitude of the
                          gradient image that passes non-maximal
                          suppression. */

   /****************************************************************************
   * Get the command line arguments.
   ****************************************************************************/
   if(argc < 5){
   fprintf(stderr,"\n<USAGE> %s image sigma tlow thigh [writedirim]\n",argv[0]);
      fprintf(stderr,"\n      image:      An image to process. Must be in ");
      fprintf(stderr,"PGM format.\n");
      fprintf(stderr,"      sigma:      Standard deviation of the gaussian");
```

```
      fprintf(stderr," blur kernel.\n");
      fprintf(stderr,"      tlow:       Fraction (0.0-1.0) of the high ");
      fprintf(stderr,"edge strength threshold.\n");
      fprintf(stderr,"      thigh:      Fraction (0.0-1.0) of the distribution");
      fprintf(stderr," of non-zero edge\n                  strengths for ");
      fprintf(stderr,"hysteresis. The fraction is used to compute\n");
      fprintf(stderr,"                  the high edge strength threshold.\n");
      fprintf(stderr,"      writedirim: Optional argument to output ");
      fprintf(stderr,"a floating point");
      fprintf(stderr," direction image.\n\n");
      exit(1);
   }

   infilename = argv[1];
   sigma = atof(argv[2]);
   tlow = atof(argv[3]);
   thigh = atof(argv[4]);

   if(argc == 6) dirfilename = infilename;
   else dirfilename = NULL;

   /****************************************************************************
   * Read in the image. This read function allocates memory for the image.
   ****************************************************************************/
   if(VERBOSE) printf("Reading the image %s.\n", infilename);
   if(read_pgm_image(infilename, &image, &rows, &cols) == 0){
      fprintf(stderr, "Error reading the input image, %s.\n", infilename);
      exit(1);
   }

   /****************************************************************************
   * Perform the edge detection. All of the work takes place here.
   ****************************************************************************/
   if(VERBOSE) printf("Starting Canny edge detection.\n");
   if(dirfilename != NULL){
      sprintf(composedfname, "%s_s_%3.2f_l_%3.2f_h_%3.2f.fim", infilename,
      sigma, tlow, thigh);
      dirfilename = composedfname;
   }
   canny(image, rows, cols, sigma, tlow, thigh, &edge, dirfilename);

   /****************************************************************************
   * Write out the edge image to a file.
   ****************************************************************************/
   sprintf(outfilename, "%s_s_%3.2f_l_%3.2f_h_%3.2f.pgm", infilename,
      sigma, tlow, thigh);
   if(VERBOSE) printf("Writing the edge iname in the file %s.\n", outfilename);
   if(write_pgm_image(outfilename, edge, rows, cols, "", 255) == 0){
      fprintf(stderr, "Error writing the edge image, %s.\n", outfilename);
      exit(1);
   }
   return(0); /* exit cleanly */
}

/*******************************************************************************
* PROCEDURE: canny
* PURPOSE: To perform canny edge detection.
* NAME: Mike Heath
* DATE: 2/15/96
```

```
*******************************************************************************/
void canny(unsigned char *image, int rows, int cols, float sigma,
         float tlow, float thigh, unsigned char **edge, char *fname)
{
   FILE *fpdir=NULL;          /* File to write the gradient image to.     */
   unsigned char *nms;        /* Points that are local maximal magnitude. */
   short int *smoothedim,     /* The image after gaussian smoothing.      */
             *delta_x,        /* The first devivative image, x-direction. */
             *delta_y,        /* The first derivative image, y-direction. */
             *magnitude;      /* The magnitude of the gadient image.      */
   int r, c, pos;
   float *dir_radians=NULL;   /* Gradient direction image.                */

   /****************************************************************************
   * Perform gaussian smoothing on the image using the input standard
   * deviation.
   ****************************************************************************/
   if(VERBOSE) printf("Smoothing the image using a gaussian kernel.\n");
   gaussian_smooth(image, rows, cols, sigma, &smoothedim);

   /****************************************************************************
   * Compute the first derivative in the x and y directions.
   ****************************************************************************/
   if(VERBOSE) printf("Computing the X and Y first derivatives.\n");
   derrivative_x_y(smoothedim, rows, cols, &delta_x, &delta_y);

   /****************************************************************************
   * This option to write out the direction of the edge gradient was added
   * to make the information available for computing an edge quality figure
   * of merit.
   ****************************************************************************/
   if(fname != NULL){
      /*************************************************************************
      * Compute the direction up the gradient, in radians that are
      * specified counteclockwise from the positive x-axis.
      *************************************************************************/
      radian_direction(delta_x, delta_y, rows, cols, &dir_radians, -1, -1);

      /*************************************************************************
      * Write the gradient direction image out to a file.
      *************************************************************************/
      if((fpdir = fopen(fname, "wb")) == NULL){
         fprintf(stderr, "Error opening the file %s for writing.\n", fname);
         exit(1);
      }
      fwrite(dir_radians, sizeof(float), rows*cols, fpdir);
      fclose(fpdir);
      free(dir_radians);
   }

   /****************************************************************************
   * Compute the magnitude of the gradient.
   ****************************************************************************/
   if(VERBOSE) printf("Computing the magnitude of the gradient.\n");
   magnitude_x_y(delta_x, delta_y, rows, cols, &magnitude);

   /****************************************************************************
   * Perform non-maximal suppression.
   ****************************************************************************/
```

```
   if(VERBOSE) printf("Doing the non-maximal suppression.\n");
   if((nms = (unsigned char *) calloc(rows*cols,sizeof(unsigned char)))==NULL){
      fprintf(stderr, "Error allocating the nms image.\n");
      exit(1);
   }

   non_max_supp(magnitude, delta_x, delta_y, rows, cols, nms);

   /****************************************************************************
   * Use hysteresis to mark the edge pixels.
   ****************************************************************************/
   if(VERBOSE) printf("Doing hysteresis thresholding.\n");
   if((*edge=(unsigned char *)calloc(rows*cols,sizeof(unsigned char))) ==NULL){
      fprintf(stderr, "Error allocating the edge image.\n");
      exit(1);
   }

   apply_hysteresis(magnitude, nms, rows, cols, tlow, thigh, *edge);

   /****************************************************************************
   * Free all of the memory that we allocated except for the edge image that
   * is still being used to store out result.
   ****************************************************************************/
   free(smoothedim);
   free(delta_x);
   free(delta_y);
   free(magnitude);
   free(nms);
}

/*******************************************************************************
* Procedure: radian_direction
* Purpose: To compute a direction of the gradient image from component dx and
* dy images. Because not all derriviatives are computed in the same way, this
* code allows for dx or dy to have been calculated in different ways.
*
* FOR X:  xdirtag = -1  for  [-1 0  1]
*         xdirtag =  1  for  [ 1 0 -1]
*
* FOR Y:  ydirtag = -1  for  [-1 0  1]'
*         ydirtag =  1  for  [ 1 0 -1]'
*
* The resulting angle is in radians measured counterclockwise from the
* xdirection. The angle points "up the gradient".
*******************************************************************************/
void radian_direction(short int *delta_x, short int *delta_y, int rows,
    int cols, float **dir_radians, int xdirtag, int ydirtag)
{
   int r, c, pos;
   float *dirim=NULL;
   double dx, dy;

   /****************************************************************************
   * Allocate an image to store the direction of the gradient.
   ****************************************************************************/
   if((dirim = (float *) calloc(rows*cols, sizeof(float))) == NULL){
      fprintf(stderr, "Error allocating the gradient direction image.\n");
      exit(1);
   }
```

```
   *dir_radians = dirim;

   for(r=0,pos=0;r<rows;r++){
      for(c=0;c<cols;c++,pos++){
         dx = (double)delta_x[pos];
         dy = (double)delta_y[pos];

         if(xdirtag == 1) dx = -dx;
         if(ydirtag == -1) dy = -dy;

         dirim[pos] = (float)angle_radians(dx, dy);
      }
   }
}

/*******************************************************************************
* FUNCTION: angle_radians
* PURPOSE: This procedure computes the angle of a vector with components x and
* y. It returns this angle in radians with the answer being in the range
* 0 <= angle <2*PI.
*******************************************************************************/
double angle_radians(double x, double y)
{
   double xu, yu, ang;

   xu = fabs(x);
   yu = fabs(y);

   if((xu == 0) && (yu == 0)) return(0);

   ang = atan(yu/xu);

   if(x >= 0){
      if(y >= 0) return(ang);
      else return(2*M_PI - ang);
   }
   else{
      if(y >= 0) return(M_PI - ang);
      else return(M_PI + ang);
   }
}

/*******************************************************************************
* PROCEDURE: magnitude_x_y
* PURPOSE: Compute the magnitude of the gradient. This is the square root of
* the sum of the squared derivative values.
* NAME: Mike Heath
* DATE: 2/15/96
*******************************************************************************/
void magnitude_x_y(short int *delta_x, short int *delta_y, int rows, int cols,
        short int **magnitude)
{
   int r, c, pos, sq1, sq2;

   /****************************************************************************
   * Allocate an image to store the magnitude of the gradient.
   ****************************************************************************/
   if((*magnitude = (short *) calloc(rows*cols, sizeof(short))) == NULL){
      fprintf(stderr, "Error allocating the magnitude image.\n");
```

```
      exit(1);
   }

   for(r=0,pos=0;r<rows;r++){
      for(c=0;c<cols;c++,pos++){
         sq1 = (int)delta_x[pos] * (int)delta_x[pos];
         sq2 = (int)delta_y[pos] * (int)delta_y[pos];
         (*magnitude)[pos] = (short)(0.5 + sqrt((float)sq1 + (float)sq2));
      }
   }

}

/*******************************************************************************
* PROCEDURE: derrivative_x_y
* PURPOSE: Compute the first derivative of the image in both the x any y
* directions. The differential filters that are used are:
*
*                                          -1
*         dx =  -1 0 +1     and       dy =  0
*                                          +1
*
* NAME: Mike Heath
* DATE: 2/15/96
*******************************************************************************/
void derrivative_x_y(short int *smoothedim, int rows, int cols,
        short int **delta_x, short int **delta_y)
{
   int r, c, pos;

   /****************************************************************************
   * Allocate images to store the derivatives.
   ****************************************************************************/
   if(((*delta_x) = (short *) calloc(rows*cols, sizeof(short))) == NULL){
      fprintf(stderr, "Error allocating the delta_x image.\n");
      exit(1);
   }
   if(((*delta_y) = (short *) calloc(rows*cols, sizeof(short))) == NULL){
      fprintf(stderr, "Error allocating the delta_x image.\n");
      exit(1);
   }

   /****************************************************************************
   * Compute the x-derivative. Adjust the derivative at the borders to avoid
   * losing pixels.
   ****************************************************************************/
   if(VERBOSE) printf("   Computing the X-direction derivative.\n");
   for(r=0;r<rows;r++){
      pos = r * cols;
      (*delta_x)[pos] = smoothedim[pos+1] - smoothedim[pos];
      pos++;
      for(c=1;c<(cols-1);c++,pos++){
         (*delta_x)[pos] = smoothedim[pos+1] - smoothedim[pos-1];
      }
      (*delta_x)[pos] = smoothedim[pos] - smoothedim[pos-1];
   }

   /****************************************************************************
   * Compute the y-derivative. Adjust the derivative at the borders to avoid
```

```
   * losing pixels.
   ***************************************************************************/
   if(VERBOSE) printf("   Computing the Y-direction derivative.\n");
   for(c=0;c<cols;c++){
      pos = c;
      (*delta_y)[pos] = smoothedim[pos+cols] - smoothedim[pos];
      pos += cols;
      for(r=1;r<(rows-1);r++,pos+=cols){
         (*delta_y)[pos] = smoothedim[pos+cols] - smoothedim[pos-cols];
      }
      (*delta_y)[pos] = smoothedim[pos] - smoothedim[pos-cols];
   }
}

/*******************************************************************************
* PROCEDURE: gaussian_smooth
* PURPOSE: Blur an image with a gaussian filter.
* NAME: Mike Heath
* DATE: 2/15/96
*******************************************************************************/
void gaussian_smooth(unsigned char *image, int rows, int cols, float sigma,
        short int **smoothedim)
{
   int r, c, rr, cc,     /* Counter variables. */
      windowsize,        /* Dimension of the gaussian kernel. */
      center;            /* Half of the windowsize. */
   float *tempim,        /* Buffer for separable filter gaussian smoothing. */
         *kernel,        /* A one dimensional gaussian kernel. */
         dot,            /* Dot product summing variable. */
         sum;            /* Sum of the kernel weights variable. */

   /****************************************************************************
   * Create a 1-dimensional gaussian smoothing kernel.
   ****************************************************************************/
   if(VERBOSE) printf("   Computing the gaussian smoothing kernel.\n");
   make_gaussian_kernel(sigma, &kernel, &windowsize);
   center = windowsize / 2;

   /****************************************************************************
   * Allocate a temporary buffer image and the smoothed image.
   ****************************************************************************/
   if((tempim = (float *) calloc(rows*cols, sizeof(float))) == NULL){
      fprintf(stderr, "Error allocating the buffer image.\n");
      exit(1);
   }
   if(((*smoothedim) = (short int *) calloc(rows*cols,
         sizeof(short int))) == NULL){
      fprintf(stderr, "Error allocating the smoothed image.\n");
      exit(1);
   }

   /****************************************************************************
   * Blur in the x - direction.
   ****************************************************************************/
   if(VERBOSE) printf("   Bluring the image in the X-direction.\n");
   for(r=0;r<rows;r++){
      for(c=0;c<cols;c++){
         dot = 0.0;
         sum = 0.0;
```

```
         for(cc=(-center);cc<=center;cc++){
            if(((c+cc) >= 0) && ((c+cc) < cols)){
               dot += (float)image[r*cols+(c+cc)] * kernel[center+cc];
               sum += kernel[center+cc];
            }
         }
         tempim[r*cols+c] = dot/sum;
      }
   }

   /****************************************************************************
   * Blur in the y - direction.
   ****************************************************************************/
   if(VERBOSE) printf("   Bluring the image in the Y-direction.\n");
   for(c=0;c<cols;c++){
      for(r=0;r<rows;r++){
         sum = 0.0;
         dot = 0.0;
         for(rr=(-center);rr<=center;rr++){
            if(((r+rr) >= 0) && ((r+rr) < rows)){
               dot += tempim[(r+rr)*cols+c] * kernel[center+rr];
               sum += kernel[center+rr];
            }
         }
         (*smoothedim)[r*cols+c] = (short int)(dot*BOOSTBLURFACTOR/sum + 0.5);
      }
   }

   free(tempim);
   free(kernel);
}

/*******************************************************************************
* PROCEDURE: make_gaussian_kernel
* PURPOSE: Create a one dimensional gaussian kernel.
* NAME: Mike Heath
* DATE: 2/15/96
*******************************************************************************/
void make_gaussian_kernel(float sigma, float **kernel, int *windowsize)
{
   int i, center;
   float x, fx, sum=0.0;

   *windowsize = 1 + 2 * ceil(2.5 * sigma);
   center = (*windowsize) / 2;

   if(VERBOSE) printf("      The kernel has %d elements.\n", *windowsize);
   if((*kernel = (float *) calloc((*windowsize), sizeof(float))) == NULL){
      fprintf(stderr, "Error callocing the gaussian kernel array.\n");
      exit(1);
   }

   for(i=0;i<(*windowsize);i++){
      x = (float)(i - center);
      fx = pow(2.71828, -0.5*x*x/(sigma*sigma)) / (sigma * sqrt(6.2831853));
      (*kernel)[i] = fx;
      sum += fx;
   }
```

```
   for(i=0;i<(*windowsize);i++) (*kernel)[i] /= sum;

   if(VERBOSE){
      printf("The filter coefficients are:\n");
      for(i=0;i<(*windowsize);i++)
         printf("kernel[%d] = %f\n", i, (*kernel)[i]);
   }
}



/*******************************************************************************
* PROCEDURE: follow_edges
* PURPOSE: This procedure edges is a recursive routine that traces edgs along
* all paths whose magnitude values remain above some specifyable lower
* threshhold.
* NAME: Mike Heath
* DATE: 2/15/96
*******************************************************************************/
follow_edges(unsigned char *edgemapptr, short *edgemagptr, short lowval,
   int cols)
{
   short *tempmagptr;
   unsigned char *tempmapptr;
   int i;
   float thethresh;
   int x[8] = {1,1,0,-1,-1,-1,0,1},
       y[8] = {0,1,1,1,0,-1,-1,-1};

   for(i=0;i<8;i++){
      tempmapptr = edgemapptr - y[i]*cols + x[i];
      tempmagptr = edgemagptr - y[i]*cols + x[i];

      if((*tempmapptr == POSSIBLE_EDGE) && (*tempmagptr > lowval)){
         *tempmapptr = (unsigned char) EDGE;
         follow_edges(tempmapptr,tempmagptr, lowval, cols);
      }
   }
}

/*******************************************************************************
* PROCEDURE: apply_hysteresis
* PURPOSE: This routine finds edges that are above some high threshhold or
* are connected to a high pixel by a path of pixels greater than a low
* threshold.
* NAME: Mike Heath
* DATE: 2/15/96
*******************************************************************************/
void apply_hysteresis(short int *mag, unsigned char *nms, int rows, int cols,
      float tlow, float thigh, unsigned char *edge)
{
   int r, c, pos, numedges, lowcount, highcount, lowthreshold, highthreshold,
       i, hist[32768], rr, cc;
   short int maximum_mag, sumpix;

   /****************************************************************************
   * Initialize the edge map to possible edges everywhere the non-maximal
   * suppression suggested there could be an edge except for the border. At
   * the border we say there can not be an edge because it makes the
```

```
* follow_edges algorithm more efficient to not worry about tracking an
* edge off the side of the image.
****************************************************************************/
for(r=0,pos=0;r<rows;r++){
   for(c=0;c<cols;c++,pos++){
    if(nms[pos] == POSSIBLE_EDGE) edge[pos] = POSSIBLE_EDGE;
    else edge[pos] = NOEDGE;
   }
}

for(r=0,pos=0;r<rows;r++,pos+=cols){
   edge[pos] = NOEDGE;
   edge[pos+cols-1] = NOEDGE;
}
pos = (rows-1) * cols;
for(c=0;c<cols;c++,pos++){
   edge[c] = NOEDGE;
   edge[pos] = NOEDGE;
}

/****************************************************************************
* Compute the histogram of the magnitude image. Then use the histogram to
* compute hysteresis thresholds.
****************************************************************************/
for(r=0;r<32768;r++) hist[r] = 0;
for(r=0,pos=0;r<rows;r++){
   for(c=0;c<cols;c++,pos++){
    if(edge[pos] == POSSIBLE_EDGE) hist[mag[pos]]++;
   }
}

/****************************************************************************
* Compute the number of pixels that passed the nonmaximal suppression.
****************************************************************************/
for(r=1,numedges=0;r<32768;r++){
   if(hist[r] != 0) maximum_mag = r;
   numedges += hist[r];
}

highcount = (int)(numedges * thigh + 0.5);

/****************************************************************************
* Compute the high threshold value as the (100 * thigh) percentage point
* in the magnitude of the gradient histogram of all the pixels that passes
* non-maximal suppression. Then calculate the low threshold as a fraction
* of the computed high threshold value. John Canny said in his paper
* "A Computational Approach to Edge Detection" that "The ratio of the
* high to low threshold in the implementation is in the range two or three
* to one." That means that in terms of this implementation, we should
* choose tlow ~= 0.5 or 0.33333.
****************************************************************************/
r = 1;
numedges = hist[1];
while((r<(maximum_mag-1)) && (numedges < highcount)){
   r++;
   numedges += hist[r];
}
highthreshold = r;
lowthreshold = (int)(highthreshold * tlow + 0.5);
```

```
   if(VERBOSE){
      printf("The input low and high fractions of %f and %f computed to\n",
       tlow, thigh);
      printf("magnitude of the gradient threshold values of: %d %d\n",
       lowthreshold, highthreshold);
   }

   /****************************************************************************
   * This loop looks for pixels above the highthreshold to locate edges and
   * then calls follow_edges to continue the edge.
   ****************************************************************************/
   for(r=0,pos=0;r<rows;r++){
      for(c=0;c<cols;c++,pos++){
       if((edge[pos] == POSSIBLE_EDGE) && (mag[pos] >= highthreshold)){
            edge[pos] = EDGE;
            follow_edges((edge+pos), (mag+pos), lowthreshold, cols);
       }
      }
   }

   /****************************************************************************
   * Set all the remaining possible edges to non-edges.
   ****************************************************************************/
   for(r=0,pos=0;r<rows;r++){
      for(c=0;c<cols;c++,pos++) if(edge[pos] != EDGE) edge[pos] = NOEDGE;
   }
}

/*******************************************************************************
* PROCEDURE: non_max_supp
* PURPOSE: This routine applies non-maximal suppression to the magnitude of
* the gradient image.
* NAME: Mike Heath
* DATE: 2/15/96
*******************************************************************************/
non_max_supp(short *mag, short *gradx, short *grady, int nrows, int ncols,
    unsigned char *result)
{
    int rowcount, colcount,count;
    short *magrowptr,*magptr;
    short *gxrowptr,*gxptr;
    short *gyrowptr,*gyptr,z1,z2;
    short m00,gx,gy;
    float mag1,mag2,xperp,yperp;
    unsigned char *resultrowptr, *resultptr;


   /****************************************************************************
   * Zero the edges of the result image.
   ****************************************************************************/
    for(count=0,resultrowptr=result,resultptr=result+ncols*(nrows-1);
        count<ncols; resultptr++,resultrowptr++,count++){
        *resultrowptr = *resultptr = (unsigned char) 0;
    }

    for(count=0,resultptr=result,resultrowptr=result+ncols-1;
        count<nrows; count++,resultptr+=ncols,resultrowptr+=ncols){
        *resultptr = *resultrowptr = (unsigned char) 0;
```

```
 }

/*******************************************************************************
* Suppress non-maximum points.
*******************************************************************************/
for(rowcount=1,magrowptr=mag+ncols+1,gxrowptr=gradx+ncols+1,
   gyrowptr=grady+ncols+1,resultrowptr=result+ncols+1;
   rowcount<=nrows-2;      /* bug fix 3/29/17, RD */
   rowcount++,magrowptr+=ncols,gyrowptr+=ncols,gxrowptr+=ncols,
   resultrowptr+=ncols){
   for(colcount=1,magptr=magrowptr,gxptr=gxrowptr,gyptr=gyrowptr,
      resultptr=resultrowptr;colcount<=ncols-2;   /* bug fix 3/29/17, RD */
      colcount++,magptr++,gxptr++,gyptr++,resultptr++){
      m00 = *magptr;
      if(m00 == 0){
         *resultptr = (unsigned char) NOEDGE;
      }
      else{
         xperp = -(gx = *gxptr)/((float)m00);
         yperp = (gy = *gyptr)/((float)m00);
      }

      if(gx >= 0){
         if(gy >= 0){
                 if (gx >= gy)
                 {
                     /* 111 */
                     /* Left point */
                     z1 = *(magptr - 1);
                     z2 = *(magptr - ncols - 1);

                     mag1 = (m00 - z1)*xperp + (z2 - z1)*yperp;

                     /* Right point */
                     z1 = *(magptr + 1);
                     z2 = *(magptr + ncols + 1);

                     mag2 = (m00 - z1)*xperp + (z2 - z1)*yperp;
                 }
                 else
                 {
                     /* 110 */
                     /* Left point */
                     z1 = *(magptr - ncols);
                     z2 = *(magptr - ncols - 1);

                     mag1 = (z1 - z2)*xperp + (z1 - m00)*yperp;

                     /* Right point */
                     z1 = *(magptr + ncols);
                     z2 = *(magptr + ncols + 1);

                     mag2 = (z1 - z2)*xperp + (z1 - m00)*yperp;
                 }
             }
             else
             {
                 if (gx >= -gy)
                 {
```

```
                /* 101 */
                /* Left point */
                z1 = *(magptr - 1);
                z2 = *(magptr + ncols - 1);

                mag1 = (m00 - z1)*xperp + (z1 - z2)*yperp;

                /* Right point */
                z1 = *(magptr + 1);
                z2 = *(magptr - ncols + 1);

                mag2 = (m00 - z1)*xperp + (z1 - z2)*yperp;
            }
            else
            {
                /* 100 */
                /* Left point */
                z1 = *(magptr + ncols);
                z2 = *(magptr + ncols - 1);

                mag1 = (z1 - z2)*xperp + (m00 - z1)*yperp;

                /* Right point */
                z1 = *(magptr - ncols);
                z2 = *(magptr - ncols + 1);

                mag2 = (z1 - z2)*xperp  + (m00 - z1)*yperp;
            }
        }
    }
    else
    {
        if ((gy = *gyptr) >= 0)
        {
            if (-gx >= gy)
            {
                /* 011 */
                /* Left point */
                z1 = *(magptr + 1);
                z2 = *(magptr - ncols + 1);

                mag1 = (z1 - m00)*xperp + (z2 - z1)*yperp;

                /* Right point */
                z1 = *(magptr - 1);
                z2 = *(magptr + ncols - 1);

                mag2 = (z1 - m00)*xperp + (z2 - z1)*yperp;
            }
            else
            {
                /* 010 */
                /* Left point */
                z1 = *(magptr - ncols);
                z2 = *(magptr - ncols + 1);

                mag1 = (z2 - z1)*xperp + (z1 - m00)*yperp;

                /* Right point */
```

```
                z1 = *(magptr + ncols);
                z2 = *(magptr + ncols - 1);

                mag2 = (z2 - z1)*xperp + (z1 - m00)*yperp;
            }
        }
        else
        {
            if (-gx > -gy)
            {
                /* 001 */
                /* Left point */
                z1 = *(magptr + 1);
                z2 = *(magptr + ncols + 1);

                mag1 = (z1 - m00)*xperp + (z1 - z2)*yperp;

                /* Right point */
                z1 = *(magptr - 1);
                z2 = *(magptr - ncols - 1);

                mag2 = (z1 - m00)*xperp + (z1 - z2)*yperp;
            }
            else
            {
                /* 000 */
                /* Left point */
                z1 = *(magptr + ncols);
                z2 = *(magptr + ncols + 1);

                mag1 = (z2 - z1)*xperp + (m00 - z1)*yperp;

                /* Right point */
                z1 = *(magptr - ncols);
                z2 = *(magptr - ncols - 1);

                mag2 = (z2 - z1)*xperp + (m00 - z1)*yperp;
            }
        }
    }

    /* Now determine if the current point is a maximum point */

    if ((mag1 > 0.0) || (mag2 > 0.0))
    {
        *resultptr = (unsigned char) NOEDGE;
    }
    else
    {
        if (mag2 == 0.0)
            *resultptr = (unsigned char) NOEDGE;
        else
            *resultptr = (unsigned char) POSSIBLE_EDGE;
    }
  }
 }
}
```

```c
/*******************************************************************************
* Function: read_pgm_image
* Purpose: This function reads in an image in PGM format. The image can be
* read in from either a file or from standard input. The image is only read
* from standard input when infilename = NULL. Because the PGM format includes
* the number of columns and the number of rows in the image, these are read
* from the file. Memory to store the image is allocated in this function.
* All comments in the header are discarded in the process of reading the
* image. Upon failure, this function returns 0, upon sucess it returns 1.
*******************************************************************************/
int read_pgm_image(char *infilename, unsigned char **image, int *rows,
    int *cols)
{
   FILE *fp;
   char buf[71];

   /***************************************************************************
   * Open the input image file for reading if a filename was given. If no
   * filename was provided, set fp to read from standard input.
   ***************************************************************************/
   if(infilename == NULL) fp = stdin;
   else{
      if((fp = fopen(infilename, "r")) == NULL){
         fprintf(stderr, "Error reading the file %s in read_pgm_image().\n",
            infilename);
         return(0);
      }
   }

   /***************************************************************************
   * Verify that the image is in PGM format, read in the number of columns
   * and rows in the image and scan past all of the header information.
   ***************************************************************************/
   fgets(buf, 70, fp);
   if(strncmp(buf,"P5",2) != 0){
      fprintf(stderr, "The file %s is not in PGM format in ", infilename);
      fprintf(stderr, "read_pgm_image().\n");
      if(fp != stdin) fclose(fp);
      return(0);
   }
   do{ fgets(buf, 70, fp); }while(buf[0] == '#');  /* skip all comment lines */
   sscanf(buf, "%d %d", cols, rows);
   do{ fgets(buf, 70, fp); }while(buf[0] == '#');  /* skip all comment lines */

   /***************************************************************************
   * Allocate memory to store the image then read the image from the file.
   ***************************************************************************/
   if(((*image) = (unsigned char *) malloc((*rows)*(*cols))) == NULL){
      fprintf(stderr, "Memory allocation failure in read_pgm_image().\n");
      if(fp != stdin) fclose(fp);
      return(0);
   }
   if((*rows) != fread((*image), (*cols), (*rows), fp)){
      fprintf(stderr, "Error reading the image data in read_pgm_image().\n");
      if(fp != stdin) fclose(fp);
      free((*image));
      return(0);
   }
```

```
   if(fp != stdin) fclose(fp);
   return(1);
}

/*******************************************************************************
* Function: write_pgm_image
* Purpose: This function writes an image in PGM format. The file is either
* written to the file specified by outfilename or to standard output if
* outfilename = NULL. A comment can be written to the header if coment != NULL.
*******************************************************************************/
int write_pgm_image(char *outfilename, unsigned char *image, int rows,
    int cols, char *comment, int maxval)
{
   FILE *fp;

   /***************************************************************************
   * Open the output image file for writing if a filename was given. If no
   * filename was provided, set fp to write to standard output.
   ***************************************************************************/
   if(outfilename == NULL) fp = stdout;
   else{
      if((fp = fopen(outfilename, "w")) == NULL){
         fprintf(stderr, "Error writing the file %s in write_pgm_image().\n",
            outfilename);
         return(0);
      }
   }

   /***************************************************************************
   * Write the header information to the PGM file.
   ***************************************************************************/
   fprintf(fp, "P5\n%d %d\n", cols, rows);
   if(comment != NULL)
      if(strlen(comment) <= 70) fprintf(fp, "# %s\n", comment);
   fprintf(fp, "%d\n", maxval);

   /***************************************************************************
   * Write the image data to the file.
   ***************************************************************************/
   if(rows != fwrite(image, cols, rows, fp)){
      fprintf(stderr, "Error writing the image data in write_pgm_image().\n");
      if(fp != stdout) fclose(fp);
      return(0);
   }

   if(fp != stdout) fclose(fp);
   return(1);
}

/*******************************************************************************
* Function: read_ppm_image
* Purpose: This function reads in an image in PPM format. The image can be
* read in from either a file or from standard input. The image is only read
* from standard input when infilename = NULL. Because the PPM format includes
* the number of columns and the number of rows in the image, these are read
* from the file. Memory to store the image is allocated in this function.
* All comments in the header are discarded in the process of reading the
* image. Upon failure, this function returns 0, upon sucess it returns 1.
*******************************************************************************/
```

```
int read_ppm_image(char *infilename, unsigned char **image_red,
    unsigned char **image_grn, unsigned char **image_blu, int *rows,
    int *cols)
{
   FILE *fp;
   char buf[71];
   int p, size;

   /***************************************************************************
   * Open the input image file for reading if a filename was given. If no
   * filename was provided, set fp to read from standard input.
   ***************************************************************************/
   if(infilename == NULL) fp = stdin;
   else{
      if((fp = fopen(infilename, "r")) == NULL){
         fprintf(stderr, "Error reading the file %s in read_ppm_image().\n",
            infilename);
         return(0);
      }
   }

   /***************************************************************************
   * Verify that the image is in PPM format, read in the number of columns
   * and rows in the image and scan past all of the header information.
   ***************************************************************************/
   fgets(buf, 70, fp);
   if(strncmp(buf,"P6",2) != 0){
      fprintf(stderr, "The file %s is not in PPM format in ", infilename);
      fprintf(stderr, "read_ppm_image().\n");
      if(fp != stdin) fclose(fp);
      return(0);
   }
   do{ fgets(buf, 70, fp); }while(buf[0] == '#');  /* skip all comment lines */
   sscanf(buf, "%d %d", cols, rows);
   do{ fgets(buf, 70, fp); }while(buf[0] == '#');  /* skip all comment lines */

   /***************************************************************************
   * Allocate memory to store the image then read the image from the file.
   ***************************************************************************/
   if(((*image_red) = (unsigned char *) malloc((*rows)*(*cols))) == NULL){
      fprintf(stderr, "Memory allocation failure in read_ppm_image().\n");
      if(fp != stdin) fclose(fp);
      return(0);
   }
   if(((*image_grn) = (unsigned char *) malloc((*rows)*(*cols))) == NULL){
      fprintf(stderr, "Memory allocation failure in read_ppm_image().\n");
      if(fp != stdin) fclose(fp);
      return(0);
   }
   if(((*image_blu) = (unsigned char *) malloc((*rows)*(*cols))) == NULL){
      fprintf(stderr, "Memory allocation failure in read_ppm_image().\n");
      if(fp != stdin) fclose(fp);
      return(0);
   }

   size = (*rows)*(*cols);
   for(p=0;p<size;p++){
      (*image_red)[p] = (unsigned char)fgetc(fp);
      (*image_grn)[p] = (unsigned char)fgetc(fp);
```

```
      (*image_blu)[p] = (unsigned char)fgetc(fp);
   }

   if(fp != stdin) fclose(fp);
   return(1);
}

/*******************************************************************************
* Function: write_ppm_image
* Purpose: This function writes an image in PPM format. The file is either
* written to the file specified by outfilename or to standard output if
* outfilename = NULL. A comment can be written to the header if coment != NULL.
*******************************************************************************/
int write_ppm_image(char *outfilename, unsigned char *image_red,
    unsigned char *image_grn, unsigned char *image_blu, int rows,
    int cols, char *comment, int maxval)
{
   FILE *fp;
   long size, p;

   /****************************************************************************
   * Open the output image file for writing if a filename was given. If no
   * filename was provided, set fp to write to standard output.
   ****************************************************************************/
   if(outfilename == NULL) fp = stdout;
   else{
      if((fp = fopen(outfilename, "w")) == NULL){
         fprintf(stderr, "Error writing the file %s in write_pgm_image().\n",
            outfilename);
         return(0);
      }
   }

   /****************************************************************************
   * Write the header information to the PGM file.
   ****************************************************************************/
   fprintf(fp, "P6\n%d %d\n", cols, rows);
   if(comment != NULL)
      if(strlen(comment) <= 70) fprintf(fp, "# %s\n", comment);
   fprintf(fp, "%d\n", maxval);

   /****************************************************************************
   * Write the image data to the file.
   ****************************************************************************/
   size = (long)rows * (long)cols;
   for(p=0;p<size;p++){      /* Write the image in pixel interleaved format. */
      fputc(image_red[p], fp);
      fputc(image_grn[p], fp);
      fputc(image_blu[p], fp);
   }

   if(fp != stdout) fclose(fp);
   return(1);
}
```