CS664 Homework Assignment 2 Results

Christopher Kline (ckline@cs)
Daniel DiPasquo (dipasquo@cs)

April 5, 1996

Contents

1 Two Dimensional Motion Fields **2**
1.1 Implementation 2
1.1.1 Normalized Correlation 2
1.1.2 Rank Transform 2
1.2 Interface 3
1.3 Examples 3
1.4 Suggestions for Improvement 4

2 Snakes: Active Contour Models **6**
2.1 Implementation 6
2.2 Examples 6
2.3 Suggestions for improvement 7
2.4 Interface 8

3 Snakes and Motion Fields Together for Motion Tracking **11**

4 Snakes Code **12**

5 Motion Code **20**

1 Two Dimensional Motion Fields

1.1 Implementation

Two algorithms for computing two dimensional motion fields were implemented. Both are area-based approaches that correlate neighborhoods of pixels rather than single pixels. The first, normalized correlation, is based on statistical methods. The second implementation is of the rank transform method described by Zabih and Woodfill.

1.1.1 Normalized Correlation

We implement normalized correlation as described in lecture. Dynamic programming techniques were used to make computation of image statistics more efficient.

Statistical information for the (chronologically) first image (i.e. mean and standard deviation) is computed only once for each match window. No savings can be obtained in these computations. However, match windows are computed repeatedly in the second image because search windows overlap. The calculations are memoized, so that repeated determinations of mean and standard deviation values amounts to a table lookup.

The function *normCorrelation* computes dx and dy disparity images given two input images. This function only makes external function calls when computing values for mean and standard deviation by "brute force" (i.e. rather than a lookup of a memoized value). Limited use of external functions minimizes overhead from function calls.

1.1.2 Rank Transform

We implement an algorithm that determines correspondence between rank transformed images, that is, a non-parametric measure of local intensity described by Zabih and Woodfill. The two steps in computing rank transform correlation are building the rank transformed images and performing the correlation on them.

The first step is to perform create rank transformed images from the input images. This is handled by the *createRankTransformedImage* function. Since the runtime of this operation grows only with the size of the image and the size of the rank neighborhood, the computation of rank transform is done entirely by brute force without too much inefficiency.

The second step of the rank transform algorithm is actually determining dx and dy disparities by computing L_1 correlation between the rank transformed images, that is, minimizing the absolute differences between pixels in the match window. A "sliding window" approach is used to simplify the computation. The function *rankTransform* performs the correlation and creates the disparity images from the input images, calling *createRankTransformedImages* to, you guessed it, create the rank transformed images.

1.2 Interface

The `motion` program is executed from the command line:

```
% motion image1 image2 hdisp vdisp hconv vconv [hrank] [vrank]
```

`image1` is the name of the file containing the first input image

`image2` is the name of the file containing the second input image

`hdisp` is the maximum horizontal disparity

`vdisp` is the maximum vertical disparity

`hconv` is the horizontal convolution neighborhood

`vconv` is the vertical convolution neighborhood

`hrank` is the horizontal rank neighborhood

`vrank` is the vertical rank neighborhood

If arguments are given for `hrank` and `vrank`, the disparity images are created using the rank transform algorithm. If these arguments are left out, the normalized correlation algorithm is used.

1.3 Examples

The *ltree.pgm* and *rtree.pgm* images show in Figure 1 were used so show sample output. They provide a good reference, because we know what the disparity images should look like from our results from using Barnard's algorithm in problem set 1.

Figure 2 shows the results of using normalized correlation to compute dx and dy disparity images on the tree stereo pair. The dx image is clearly a similar results that was obtained using Barnard. The dy disparity image is not clear, but can be seen better by running the normalized correlation algorithm on the tree stereo images rotated 90 degrees clockwise. The third image shows the results from doing so.

Figure 3 shows the results of computing dx and dy disparity images on the tree stereo pair using rank transform correlation. The resulting images are similar to the results obtained using normalized correlation. Figure 4 shows the rank transformation of the right image of the tree stereo pair.

Finally, Figure 5 summarizes the parameters that were used to generate the example images. Horizontal and vertical disparities were set to 8 pixels, and horizontal and vertical convolution neighborhoods set to 4 pixels for both algorithms. In addition, the rank transform correlation was run with a rank neighborhood of 4 pixels.

1.4 Suggestions for Improvement

There are two areas in which I would like to improve the code that implements the normalized correlation and rank transform algorithms.

- Both correlation methods iterate over every pixel in the image and convolute the match window over the area of a search window. Much of the code necessary to do this could have been shared. However, because of the differences in the algorithm, a sea of *if-then* conditionals would be necessary to combine the code. I would like to find a more elegant solution to this problem.
- Both functions, *normCorrelation* and *rankTransform* are very long and uncouth. I would like to break the function down, but would need to analyze the effects of the overhead of function calls.

Figure 1: The tree stereo pair

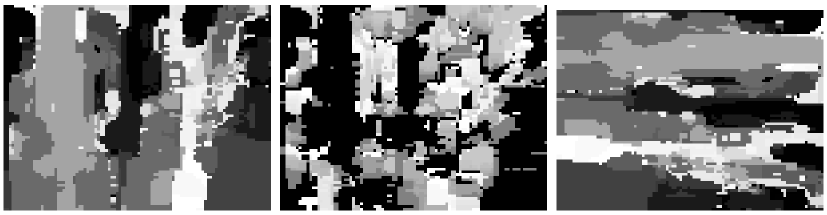


Figure 2: The dx, dy, and $dy90$ results from normalized correlation on the tree pair

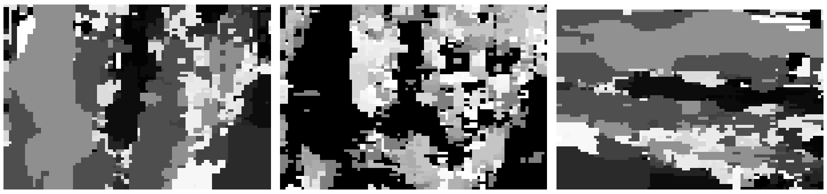


Figure 3: The dx, dy, and $dy90$ results from rank transform correlation on the tree pair

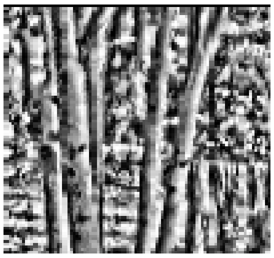


Figure 4: The rank transform of the left image from the tree pair

parameter	**horizontal**	**vertical**
disparity	8	8
convolution neighborhood	4	4
rank neighborhood	4	4

Figure 5: The values used in the computation of the images in Figures (2–4)

2 Snakes: Active Contour Models

2.1 Implementation

We have implemented a version of the "snakes" model described in Kass et al. In addition, we have used the dynamic programming techniques outlined in the Chandran paper to improve the efficiency of minimizing the discrete model of the snake and the robustness in seeking the global minimum energy.

The code is fairly straightfoward. However, since we are using a discrete as opposed to a continuous model of the snake, the handling of the endpoints required special consideration. The snake is closed in the sense that the last vertex is connected to the first, and hence the calculations for the optimal position of these vertices are interdependent.

For example, determining the minimum energy at the first vertex involves knowing the minimum energy of all the positions of the last vertex– but the energy at each position of the last vertex depends on the energy at each position of the first vertex!

While this problem can be avoided by using a continuous model and differential equations, we have chosen the discrete model for reasons stated above, and are thus forced to apply an approximation at the endpoints. In our implementation, when the first vertex queries at the energy of the last vertex, only the external energy is returned. This approximation is reasonable since the internal energy is generally much smaller than the external energy, and the first and second order continuity between the endpoints is considered at a later time when the last vertex queries the first vertex.

However, one problem in particular can arise from this approximation. Since the minimum energy of the snake is determined by building the minimum energy for each vertex in order from first to last, the minimum energy of the entire snake depends to some degree on the minimum energy of the first vertex. However, as we have described above, that energy is only an approximation, and thus the optimal positions for the first and last vertex may appear (to the snake) to be different at the time when the first vertex's minimum energy is being calculated than when the same calculation is performed for the last vertex.

The way in which this discrepancy manifests itself is that, during the minimization, the snake appears to endlessly "waffle" between two local minima. We have fixed this problem by simply checking for this behavior, and ending the minimization if it appears.

2.2 Examples

It is easily apparent from Figures (6–7) that the basic snakes algorithm is working correctly.

Figure (6) shows a sequence of frames from an animation of a snake converging on a Coke can. The Coke image is highly ideal for the snake, since it has a nearly pure light-colored background that contrasts sharply with the foreground object.

A better example of the convergence is Figure (7). In this image, the background is textured and the foreground contains multiple prominent objects. Also, the table upon which these objects rest does not contrast with them as sharply as the background of the Coke image contrasts with the Coke can. The snake converges admirably on the telephone, but notice how the snake does not tightly adhere to the right edge of the phone just under the receiver. This is due to the influence of the first and second-order terms in the internal energy, maintaining the curvature and preventing the snake from bending too sharply.

If the size of the vertex search space, m, is too great, the snake can be confused by multiple regions of high external energy. This can be observed in the "freak" image of Figure (8). In addition, the algorithm running time grows with m, so a larger search space rapidly increases the convergence time. In our experience, values of 9 or 25 are optimal for m.

2.3 Suggestions for improvement

Two ways the implementation could be improved for use on still images[1] are as follows:

- The conditions at the endpoints could be handled more accurately, perhaps by applying the continuous model at the endpoints to avoid the circular relationships that arise in our discrete approximation.

- Our implementation already allows the user to choose whether the snake is attracted to lines or edges (by setting `gradientFlag` to 0 or 1, respectively). However, it would be nice if the snake could be attracted to terminations of curvature (i.e., corners, as described by Kass). This would eliminate the problem that can be observed in Figure (6), where the snake tends to "cut corners" in order to reduce its internal energy. If the snake was attracted to corners, it would more tightly conform to the outline of the can. However, attraction to corners may be undesirable when dealing with scenes containing objects whose contours appear concave (see Figure (9)).

[1] The use of snakes in conjunction with moving images and their corresponding motion fields is covered later in this report.

2.4 Interface

The snakes program is run from the command line in the following manner:

```
% snakes pointsFile inImage outImage m alpha beta frameSpacing gradientFlag [extImage]
```

pointsFile is the name of the file containing the initial positions of the snake vertices.

inImage indicates an image upon which to superimpose the snake before writing an output image or a frame image.

outImage is the name of the image that will result from superimposing the final snake upon inImage.

m is the number of positions a vertex can move in to during each iteration; i.e., the size of the search space for each snake vertex. For example, if m $= 9$, then each vertex can move within the 3x3 pixel box surrounding it. To simplify our code, we require that m be an odd perfect square.

alpha and beta are the weights applied to the first and second-order components, respectively, of the internal energy formulation given in Equation (2) of the Kass paper.

frameSpacing indicates the number of iterations of the snake minimization algorithm that should occur before the program generates a snapshot of the snake (superimposed over inImage). I.e., if frameSpacing $= 4$ the program will output a file called frame#.pgm (where # is the frame number) every 4 times through the minimization. In this fashion, animations of the snake minimization can be constructed easily. If frameSpacing $= 0$, no frames will be generated.

extImage is an optional argument, and is the name of an image containing the external energy values for each of the (x, y) possible image positions. The image should either be a ShortImage, a DoubleImage, or a FloatImage. If this argument is not given, the program will use the inImage as the external energy image.

gradientFlag must be either 0 or 1. After the external energy image is loaded as described above, the value of this flag is checked. If it is 1, the program will generate the squared magnitude of the external energy gradient, changing the external energy such that

$$Eext(x,\ y)_{after} = |\nabla Eext(x,\ y)_{before}|^2$$

As an example of how to use this flag, the user could provide a motion field image for extImage and by setting the flag to 1, the program will automatically use the squared magnitude gradient of that motion field as the external energy.

Figure 6: Animation demonstrating snake convergence on a Coke can

frame 0


frame 1


frame 2


frame 3


frame 4


frame 5


frame 6


α	0.2
β	0.05
m	25
$E_{external}$	$\|\nabla\|^2$ of intensity

α	0.2
β	0.092
m	25
$E_{external}$	$\lvert\nabla\rvert^2$ of intensity

Figure 7: The telephone images, before and after

α	0.2
β	0.05
m	81
$E_{external}$	$\lvert\nabla\rvert^2$ of intensity

Figure 8: The freak images, before and after

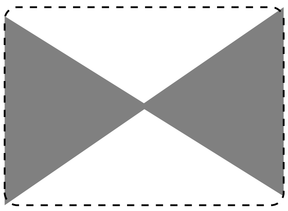


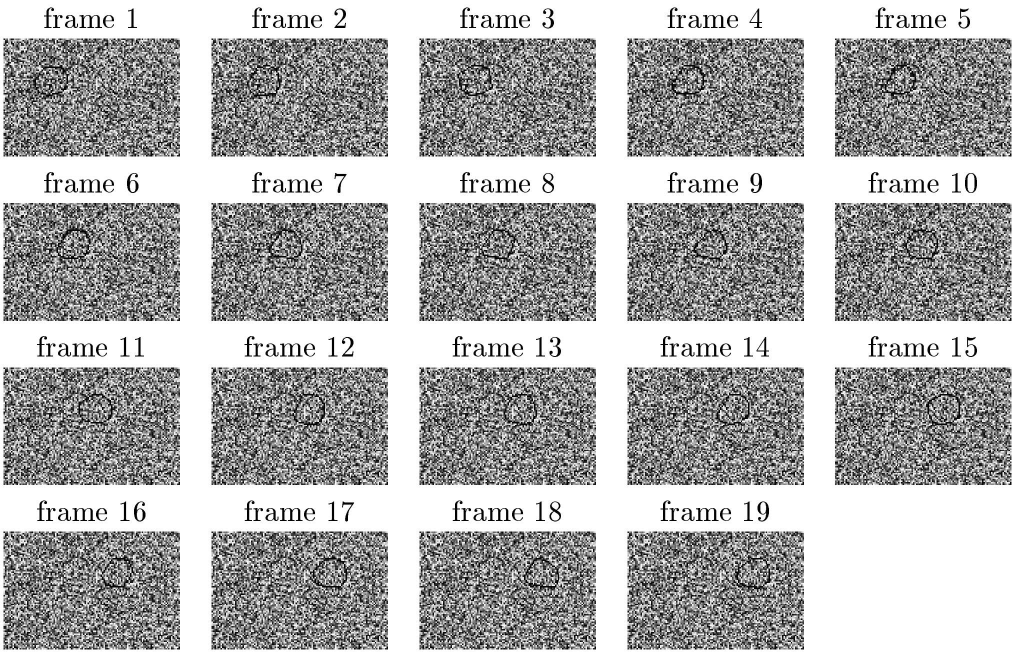
Figure 9: Simulated image demonstrating corner attraction failing on concave image contours

3 Snakes and Motion Fields Together for Motion Tracking

By using the snakes code in conjunction with the motion code, we can track a moving object. This is demonstrated in Figure (10), where a sequence of random dot stereograms contain a moving cube. Though it may be difficult to discern the motion of the cube by naked eye, if we set up the snake to use the disparities from the motion code as external energy, the snake will converge on areas of motion.

This method of motion tracking isn't foolproof, however, and breaks down when the objects move too quickly. This could be improved by imparting momentum to the snake, which would help it find the object when the object moves too quickly to be found within the vertex search space.

Figure 10: Animation demonstrating a snake tracking a moving square from motion field data



α	0.1
β	0.03
m	25
$E_{external}$	$\lvert\nabla\rvert^2$ of disparity from normalized correlation

4 Snakes Code

```
#include "image.h"
#include <math.h>
#include <limits.h>
#include <stdio.h>

#define USE_MACROS

#ifdef IMAGE_CHECKALL
static char rcsid[] = "<snakes> encountered an image check failure.";
#endif

/* SQR(i) = (i)*(i) */
#define SQR(x) ((x) * (x))

/* Returns the x or y location in the image corresponding to location p in
   the match window. Assumes that floor(a/b) == a/b if a and b are integers. */
#define xoff(x, p) (x + p%w - w/2)
#define yoff(y, p) (y + p/w - w/2)

/* The two arrays in struct layer actually represent matrices. index(a, b) gives the
   correct index into the array corresponding to matrix cell (a, b), assuming that
   the matrix is represented in row-major order */
#define index(row, column) (numPositions*row + column)

/* External energy Eext of point (pixel) at position c in the local (square matrix)
   search space. Returns a huge energy if that pixel is outside the bounds of the
   image.*/
#define Eext(point, c) ((xoff(point.x, c) < 0 || \
                         yoff(point.y, c) < 0 || \
                         xoff(point.x, c) >= imageWidth || \
                         yoff(point.y, c) >= imageHeight) \
                        ? FLT_MAX \
                        : -(imRef(externalEnergyImage, xoff(point.x, c), yoff(point.y, c))))

     /* Compute finite differences approximation to first derivative at vertex V[i] when:
   vertex V[i] is in position c
   vertex V[i+1] is in position n.

   Also take care of the special case that arise because the last vertex is
   connected to the first vertex. */
#define DERIV1TERM(i, c, n) \
     ( \
     (i < numVertices - 1) \
     ? \
     (SQR(xoff(V[i+1].x, n) - xoff(V[i].x, c)) + SQR(yoff(V[i+1].y, n) - yoff(V[i].y, c))) \
     : \
     (SQR(xoff(V[0].x, n) - xoff(V[i].x, c)) + SQR(yoff(V[0].y, n) - yoff(V[i].y, c))) \
     )

/* Compute finite differences approximation to the second derivative at vertex V[i] when:
   vertex V[i-1] is in position p,
   vertex V[i] is in position c,
   vertex V[i+1] is in position n

   Also take care of the special cases that arise because the last vertex is
   connected to the first vertex. */
#define DERIV2TERM(i, c, p, n) \
     ( \
(i == 0) \
       ? \
       (SQR(xoff(V[1].x, n) - 2*xoff(V[0].x, c) + xoff(V[numVertices-1].x, p)) + \
        SQR(yoff(V[1].y, n) - 2*yoff(V[0].y, c) + yoff(V[numVertices-1].y, p))) \
       : \
       (\
(i < numVertices - 1) \
        ? \
     (SQR(xoff(V[i+1].x, n) - 2*xoff(V[i].x, c) + xoff(V[i-1].x, p)) + \
      SQR(yoff(V[i+1].y, n) - 2*yoff(V[i].y, c) + yoff(V[i-1].y, p))) \
     : \
     (SQR(xoff(V[0].x, n) - 2*xoff(V[i].x, c) + xoff(V[i-1].x, p)) + \
      SQR(yoff(V[0].y, n) - 2*yoff(V[i].y, c) + yoff(V[i-1].y, p))) \
      ) \
      )

/* structures for snakes */
typedef struct {
  int x;
  int y;
} point;

typedef struct {
  float *E;      /* Matrix of energies for this vertex. Each row represents
                    vertex V[i] in one of its numPositions possible positions.
                    Each column represents V[i+1] in one of its numPositions
                    possible positions. The matrix is represented as an array.*/
  int *prevVp;   /* The matrix of positions of the previous vertex
                    corresponding to the values of the matrix E. */
} layer;
```

```
/* Prototypes */
void generateExternalEnergies(void);
void snakes(point V[], int numVertices, int numPositions, float alpha, float beta);
int gradientMag(FloatImage img, FloatImage *mag);
void createOutputImageFile(GrayImage im, char *filename,
                           point V[], int numVertices,
                           float alpha, float beta,
                           int numPositions);

/* Global Variables */
GrayImage inputImage;
FloatImage externalEnergyImage;

char *inputFileName;
char *outputFileName;
char *externalEnergyFileName = "NONE GIVEN";
char *pointsFileName;
int imageWidth, imageHeight;
int outputRate = 0;
int gradientFlag;

/*
  V[]          = Initial points on the snake
  numVertices  = The number of initial points on the snake
  numPositions = Number of possible positions each vertex should search (hence
                 the search window is cube of width sqrt(numPositions)).
                 numPositions must be odd.
  alpha        = Amount by which to weight the first order term
  beta         = Amount by which to weight the second order term
*/
void snakes(point V[], int numVertices, int numPositions, float alpha, float beta) {

  layer *L;
  int i, delta;
  int p, c, n; /* positions of the previous, current, and next vertex */
  int w;
  float minimumCost;
  float temp;
  float minE;
  int positionOfMin = -1;
  int tempn;
  float oldMinE = FLT_MAX; /* minimum Energy for the snake from iteration n-1*/
  float oldOldMinE = FLT_MAX; /* minimum Energy for the snake from iteration n-2*/
  float dE = FLT_MAX;      /* change in energy between cycles */
  int bestn, bestc;
  int iteration = 0;
  char frameName[60];

  /* NUMPOSITIONS must be odd and a perfect square*/
  if ((numPositions % 2 == 0) || (trunc(sqrt(numPositions)) != sqrt(numPositions))) {
    printf("numPositions must be an odd perfect square. Aborting.\n");
    exit(1);
  }

  /* Make sure the all the vertex points inside the image. */
  for (i=0; i<numVertices; i++) {
    if ((V[i].x < 0) || (V[i].x >= imageWidth) ||
        (V[i].y < 0) || (V[i].y >= imageHeight)) {
      printf("All initial snake points must be within the image. Aborting.\n");
      exit(1);
    }
  }

  /* Calculate width of search window */
  w = sqrt(numPositions);
  printf("%d possible positions for each vertex\nSearch window is %d x %d\n", numPositions, w, w);


  /* Allocate and initialize one layer for each snake vertex (except the last one) */
  L = (layer *)malloc((numVertices) * sizeof(layer));
  for (i=0; i<numVertices; i++) {
    L[i].E = (float *)malloc(numPositions*numPositions * sizeof(float));
    L[i].prevVp = (int *)malloc(numPositions*numPositions * sizeof(int));
  }

  createOutputImageFile(inputImage, "test_before.pgm", V, numVertices, alpha, beta, numPositions);

  printf("\nInitialization finished. Starting minimization:\n\n");

  /* Keep iterating the minimization until the snake finds the minimum energy in its area */
  while (dE != 0) {

    printf("Iteration %d: Minimizing Vertices\n", iteration);

  /* Cycle through each vertex (except the last), one at a
     time, calculating the minimum energy of the snake at
     each of the numPositions possible positions of vertex V[i] */

    for (i=0; i<numVertices; i++) {  /* For each vertex V[i] */
```

```
  for (c=0; c<numPositions; c++) {   /* For each possible position of this vertex */

    for (n=0; n<numPositions; n++) { /* For each possible position of the next vertex */

      /* First fill in the external energy of the point
         V[i] at its (x,y) position in the image */
      L[i].E[index(c, n)] = Eext(V[i], c);

      /* Add in the cost from the first order term */
      L[i].E[index(c, n)] += alpha * DERIV1TERM(i, c, n);

      /*
        If we're not at the first vertex, we have to consider costs from:
        1) second derivative term
        2) minimum energy of the previous vertex, depending on its position

        The plan of attack is thus:

        Loop over all possible positions of the previous vertex, and figure out
        which position of the previous vertex minimizes both the energy of the
        previous vertex AND the second derivative at the current vertex.

        Also store (in V[i].prevVp[index(c,n)]) the position of the previous
        vertex which facilitated that minimum cost, so that after we minimize all
        the snake vertex energies we can walk backward through the layers and figure out
        the correct positions of every vertex to minimize the energy of the entire
        snake. */

      minimumCost = FLT_MAX; /* set min cost to some huge value */

      for (p=0; p<numPositions; p++) { /* For all possible positions of previous vertex */

        if (i>0) {
          temp = L[i-1].E[index(p, c)];  /* Energy when the vertex V[i-1] is at position p
                                            and the vertex V[i] is at position c */
        }
        else {
          /* Special case for first vertex. Estimate the total energy as the
             external energy, since we don't have enough info at this point to
             construct the total energy (it's a circular process...) */
          temp = Eext(V[i], p);
        }

        if (beta != 0) {

          /* Add the cost of the second derivative when vertex
             V[i] is in position c, V[i-1] is in position p,
             and V[i+1] is in position n */
          temp += beta * DERIV2TERM(i, c, p, n);

        }

        if (temp < minimumCost) {
          minimumCost = temp;           /* Set the new minimum cost. */
          L[i].prevVp[index(c, n)] = p; /* Store the position of the previous vertex. */
        }

      } /* end loop over p */

      /* Store minimum cost from second order term */
      L[i].E[index(c, n)] += minimumCost;

    } /* end loop over n */

  } /* end loop over c */

} /* end loop over i*/

printf("Tracing Backwards to get new vertex positions...\n");

/* Now we walk backwards through the layers to extract out the positions
   of the snake vertices which minimize the energy of the entire snake.

   First scan the top layer and find the cell with the minimum energy value.
   This cell position (a, b) tells us the position of vertices V[n-1] (the
   last vertex).

   Then we look up the value of L[n-1].prevVp[index(a, b)], which
   contains the position of vertex V[n-2] that minimized the energy of
   vertices V[n-1].

   We do this for each vertex, so that in the end we have the positions of all
   of the vertices which minimizes the energy of the entire snake. */

minE = FLT_MAX;

for (c = 0; c<numPositions; c++) {
  for (n = 0; n<numPositions; n++) {
    if (L[numVertices-1].E[index(c, n)] < minE) {

      minE = L[numVertices-1].E[index(c, n)];
```

```
          positionOfMin = L[numVertices-1].prevVp[index(c, n)];
          bestn = n;
          bestc = c;

        }
      }
    }

    tempn = bestc;

    V[numVertices-1].x = xoff(V[numVertices-1].x, bestc);
    V[numVertices-1].y = yoff(V[numVertices-1].y, bestc);

    positionOfMin = L[numVertices-1].prevVp[index(bestc, bestn)];

    for (i = numVertices-2; i>=0; i--) {

      V[i].x = xoff(V[i].x, positionOfMin);
      V[i].y = yoff(V[i].y, positionOfMin);

      if (i !=0 ) {
        /* remember the position of V[i] */
        tempn = positionOfMin;

        /* Figure out the position of vertex V[i-1] */
        positionOfMin = L[i].prevVp[index(positionOfMin, tempn)];
      }
    }


    dE = fabs(oldMinE - minE);
    printf("(oldOldMinE = %g oldMinE = %g newMinE = %g\nChange in Energy = %g\n\n",
           oldOldMinE, oldMinE, minE, dE);

    /* Stop iterating if the snake is waffling back and forth between
       two configurations. */
    if ((oldOldMinE == minE) && (iteration != 0))
      break;

    oldOldMinE = oldMinE; /* remember the minE from the previous iteration */
    oldMinE = minE;       /* remember the minE from this iteration */

    /* Render a frame if necessary */
    if ((outputRate > 0) && (iteration % outputRate == 0)) {
      sprintf(frameName, "frame%d.pgm", iteration/outputRate);
      createOutputImageFile(inputImage, frameName, V, numVertices, alpha, beta, numPositions);
    }

    iteration++;
  }

}

int main(int argc, char **argv) {

  int numPoints;      /* number of snake vertices */
  point *snakePoints; /* the vertices */
  float alpha, beta;  /* snake parameters (weights for 1st and 2nd order terms) */
  int numPositions;

  /* Make sure we have enough input arguments. */
  if (argc < 7) {
    printf("Too few arguments given to program. Aborting.\n");
    exit(1);
  }

  /* Read in the arguments. */
  pointsFileName = argv[1];
  inputFileName = argv[2];
  outputFileName = argv[3];
  numPositions = atoi(argv[4]);
  alpha = atof(argv[5]);
  beta = atof(argv[6]);
  outputRate = atoi(argv[7]);
  gradientFlag = atoi(argv[8]);
  externalEnergyFileName = argv[9];

  /* Attempt to load in the input image. */
  inputImage = (GrayImage)imLoad(IMAGE_GRAY, inputFileName);
  if (inputImage == (GrayImage)NULL) {
    printf("Error loading input image! Aborting.\n");
    exit(1);
  }

  /* Set the dimensions of the images */
  imageWidth = (int)imGetWidth(inputImage);
  imageHeight = (int)imGetHeight(inputImage);

  /* Print debugging info about the input images */
  printf("inputImage = %ux%u\n", imageWidth, imageHeight);
```

```
  /* Load/Generate external energy for all pixels */
  generateExternalEnergies();

  /* print out the rendering frame rate */
  if (outputRate > 0) {
    printf("Outputting an image every %d iterations\n", outputRate);
  }
  else {
    outputRate = 0;
    printf("Outputting only initial and final frames.\n");
  }

  /* Read in the snake points */
  numPoints = readSnakePoints(&snakePoints);

  printf("\n\n");

  /* Minimize the snake completely */
  snakes(snakePoints, numPoints, numPositions, alpha, beta);

  /* write the output image */
  createOutputImageFile(inputImage, outputFileName, snakePoints, numPoints, alpha, beta, numPositions);

  imFree(inputImage);
  imFree(externalEnergyImage);

  return(0);
}

/*********************************************************************************/


/* Calculates the external energy Eext(i,j) for all (i, j) and
 places them in the global FloatImage externalEnergyImage */
void generateExternalEnergies(void) {

  int l;
  FloatImage temp;

  if (externalEnergyFileName == NULL) {
    /* Use input image as base for external energy */

    printf("Using %s as external energy image...\n", inputFileName);
    if ((temp = (FloatImage)imConvert(inputImage, IMAGE_FLOAT)) == NULL) {
      printf("Couldn't convert %s to a FloatImage. Aborting.\n", inputFileName);
      exit(1);
    }
  }
  else {

    /* Use the external energy image that the user specified, converting it to
       the correct type if necessary */

    printf("Using %s as external energy image...\n", externalEnergyFileName);
    l = strlen(externalEnergyFileName);

    if (externalEnergyFileName[l-2] == 'f') {
      temp = (FloatImage)imLoad(IMAGE_FLOAT, externalEnergyFileName);
      if (temp == NULL) {
        printf("Couldn't load %s. Aborting.\n", externalEnergyFileName);
        exit(1);
      }
    }
    else {
      /* Convert it to a FloatImage */

      if (externalEnergyFileName[l-2] == 's') {
        /* We have to convert it from a .psm to a .pfm */
        ShortImage im;

        printf("Converting %s to a FloatImage...\n", externalEnergyFileName);

        im = (ShortImage)imLoad(IMAGE_SHORT, externalEnergyFileName);
        temp = (FloatImage)imConvert(im, IMAGE_FLOAT);
        imFree(im);
      }
      else if (externalEnergyFileName[l-2] == 'd') {
        /* We have to convert it from a .pdm to a .pfm */
        DoubleImage im;

        printf("Converting %s to a FloatImage...\n", externalEnergyFileName);

        im = (DoubleImage)imLoad(IMAGE_DOUBLE, externalEnergyFileName);
        temp = (FloatImage)imConvert(im, IMAGE_FLOAT);
        imFree(im);
      }
      else {
        printf("%s must be a FloatImage (.pfm), ShortImage (.psm) or DoubleImage (.pdm). Aborting.\n",
               externalEnergyFileName);
        exit(1);
      }
```

```
      if (temp == NULL) {
        printf("Couldn't convert %s to a FloatImage. Aborting.\n", externalEnergyFileName);
        exit(1);
      }
    } /* end -- converting to a FloatImage */

  } /* end -- loading specified external energy image */


  if (gradientFlag != 0) {
    /* Replace the external energy data with its squared magnitude gradient */
    printf("Computing the squared magnitude gradient of the external energy.\n");
    if (gradientMag(temp, &externalEnergyImage) != 0) {
      printf("Error in gradientMag(). Aborting.\n");
      exit(1);
    }
  }
  else {
    /* Use the external energy unmodified */
    externalEnergyImage = temp;
  }

}

/* Read in the initial positions of the snake vertices from a file.
   Return the number of vertices. */
int readSnakePoints(point **V) {

  FILE *pfile;
  int x, y;
  int count = 0;
  int i = 0;

  /* yes, I know this is a slimey hack. */

  if ((pfile = fopen(pointsFileName, "r")) == NULL) {
    printf("Error opening snake points file. Aborting.\n");
    exit(1);
  }

  while (fscanf(pfile, "%d %d", &x, &y) != EOF)
    count ++;

  printf("Read %d snake points\n", count);

  fclose(pfile);

  if ((*V = (point *)malloc(count * sizeof(point))) == NULL) {
    printf("Couldn't allocate memory for snake points. Aborting.\n");
    exit(1);
  }

  if ((pfile = fopen(pointsFileName, "r")) == NULL) {
    printf("Error opening snake points file. Aborting.\n");
    exit(1);
  }

  while (fscanf(pfile, "%d %d", &x, &y) != EOF) {
    printf("x=%d y=%d\n", x, y);
    (*V)[i].x = x;
    (*V)[i].y = y;
    i++;
  }

  fclose(pfile);

  return count;

}

/* This bresenham line-drawing code snarfed from Graphics Gems Vol. 1 */
void drawLine(GrayImage im, int x1, int y1, int x2, int y2) {
  int d, x, y, ax, ay, sx, sy, dx, dy;

#define C(i, j) imRef(im, i, j)
#define cFlip(i, j) C(i, j) = (256 - C(i, j))
#define Flip(i, j) C(i, j) = 0
#define SGN(a) (((a)<0) ? -1 : 1)

  dx = x2-x1;  ax = ABS(dx)<<1;  sx = SGN(dx);
  dy = y2-y1;  ay = ABS(dy)<<1;  sy = SGN(dy);

  x = x1;
  y = y1;
  if (ax>ay) {                /* x dominant */
    d = ay-(ax>>1);
    for (;;) {
      Flip(x, y);
      if (x==x2) return;
      if (d>=0) {
```

```
        y += sy;
        d -= ax;
      }
      x += sx;
      d += ay;
    }
  }
  else {                      /* y dominant */
    d = ax-(ay>>1);
    for (;;) {
      Flip(x, y);
      if (y==y2) return;
      if (d>=0) {
        x += sx;
        d -= ay;
      }
      y += sy;
      d += ax;
    }
  }
}


void createOutputImageFile(GrayImage im, char *filename,
                           point V[], int numVertices,
                           float alpha, float beta, int numPositions) {

  int error;
  char *commentString;
  GrayImage copy;
  int i;

  copy = (GrayImage)imDup(im);
  if (copy == (GrayImage)NULL) {
    printf("Couldn't duplicate image in createOutputImageFile(). Aborting.\n");
    exit(1);
  }

  commentString = (char *)malloc(2000*sizeof(char));
  sprintf(commentString, "this file = %s\ninputImage = %s\noutputImage = %s\nexternalEnergyImage = %s\nnumPositions = %d\nalpha = %g\nbeta = %g\n",
          filename, inputFileName, outputFileName, externalEnergyFileName,
          numPositions, alpha, beta);

  imSetComment(copy, commentString);

  for (i = 0; i<numVertices-1; i++) {
    drawLine(copy, V[i].x, V[i].y, V[i+1].x, V[i+1].y);
  }
  drawLine(copy, V[numVertices-1].x, V[numVertices-1].y, V[0].x, V[0].y);

  if (imSave(copy, filename) == -1) {
    printf("Error saving image in createOutputImageFile(). Aborting.\n");
    exit(1);
  }

  imFree(copy);

 }


/*
  I stole this code from lib-support in the robotlab :

   Computes the magnitude of the gradient. Storage
   will be allocated for MAG if it is a pointer to a NULL
   FloatImage; otherwise the storage provided will be used. A non-zero value
   is returned on error -- e.g. on error allocating storage or if the result
   images passed are not the same size as the input image. */

int gradientMag(FloatImage img, FloatImage *mag) {

#define imGetStore(i)   ({ &(imRef((i), imGetXBase(i), imGetYBase(i))); })

  register float *imgp, *magp, *img_xendp;
  register float x, y;
  register long width;
  register float a, b, c, d;
  register float *img_endp;
  register long height;
  int made_mag = 0;


  if (img == (FloatImage) NULL  ||  mag == (FloatImage *) NULL)
    return -1;

  width  = imGetWidth(img);
  height = imGetHeight(img);

  if (*mag == (FloatImage) NULL) {
    if ((*mag = imNew(IMAGE_FLOAT, width, height)) == (FloatImage) NULL)
      goto gradient_error;
```

```
    made_mag = 1;
  }
  else if (imGetWidth(*mag) != width  ||  imGetHeight(*mag) != height)
    goto gradient_error;

  imgp = imGetStore(img);
  magp = imGetStore(*mag);

  for (img_endp = imgp + width*(height-1) - 1; imgp < img_endp; ) {
    a = *imgp;
    c = *(imgp+width);

    for (img_xendp = imgp + width - 1; imgp < img_xendp; ) {
      imgp++;

      b = *(imgp);
      d = *(imgp+width);

      a = d - a;
      c = b - c;
      x = a + c;
      y = a - c;

      a = b;
      c = d;

      *(magp++) = sqrt(x*x + y*y);
    }

    imgp++;                                   /* last column gets 0.0 */
    *(magp++) = 0.0;
  }

  for (img_endp += width; imgp <= img_endp; imgp++) /* last row gets 0.0 */
    *(magp++) = 0.0;

  return 0;

 gradient_error:
  if (made_mag) imFree(*mag);
  return -1;
}
```

5 Motion Code

```
#include "image.h"
#include <math.h>
#include <limits.h>

#define REPORT 1

#ifdef IMAGE_CHECKALL
static char rcsid[] = "<motion> encountered an image check failure.";
#endif

/* Read image intensities from image. */
#define I(image, i, j) imRef(image, i, j)

/* Simplify calculation of a square. */
#define SQR(x) ((x) * (x))

/* Easy access to storage and retrieval from matrices. */
#define STORE_MEAN(i, j, mean) imRef(meanMatrix, i, j) = mean
#define STORE_DEV(i, j, dev) imRef(stdDevMatrix, i, j) = dev
#define MEAN(i, j) imRef(meanMatrix, i, j)
#define STD_DEV(i, j) imRef(stdDevMatrix, i, j)

/* Global variable declarations. */

GrayImage imageA, imageB; /* Input images. */
ShortImage dxDisparityImage, dyDisparityImage; /* Output disparity images. */

char *imageAFileName;
char *imageBFileName;                     /* Command line arguments. */
int hMaxDisparity, vMaxDisparity;
int hConvNeighborhood, vConvNeighborhood;
int hRankNeighborhood = 0;
int vRankNeighborhood = 0;

/* Function prototypes. */
void normCorrelation(int imageWidth, int imageHeight);
void rankTransform(int imageWidth, int imageHeight);
float bruteCalcMatchWindowMean(GrayImage image,
                               int matchWindowTop, int matchWindowBottom,
                               int matchWindowLeft, int matchWindowRight);
float bruteCalcMatchWindowStdDev(GrayImage image, float matchWindowMean,
                                 int matchWindowTop, int matchWindowBottom,
                                 int matchWindowLeft, int matchWindowRight);
void writeDisparityImageFile(ShortImage image, char *filename,
                             int hDisparity, int vDisparity,
                             int hNeighborhood, int vNeighborhood);
GrayImage createRankTransformedImage(GrayImage image, int imageWidth,
                                     int ImageHeight);
/* Main program starts here. */

int main(int argc, char **argv) {
  /* Declaration of variables local to _main_. */

  int imageWidth, imageHeight;
  int croppedWidth, croppedHeight;
  int hOffset, vOffset;
  char *filename;

  /* Make sure we have enough input arguments. */
  if (argc < 6) {
    /* motion <imageA> <imageB> <hdisp> <vdisp> <hsearch> <vsearch>
                                                          [hrank] [vrank] */
    printf("Too few arguments given to program. Aborting.\n");
    exit(1);
  }

  /* Read in the parameters from the command line. */
  imageAFileName = argv[1]; /* Filenames for images A and B. */
  imageBFileName = argv[2];
  hMaxDisparity = atoi(argv[3]); /* Maximum horizontal and vertical */
  vMaxDisparity = atoi(argv[4]); /* disparities. */
  hConvNeighborhood = atoi(argv[5]); /* Dimensions of convolution */
  vConvNeighborhood = atoi(argv[6]); /* neighborhood. */

  /* If the user has given additional arguments, the dimensions of the rank
     neighborhood, we do rank transform, otherwise we use normalized
     correlation. */

  if (argc > 6) {
    hRankNeighborhood = atoi(argv[7]);
    vRankNeighborhood = atoi(argv[8]);
  }

  /* Let the user know what the program _thinks_ it's doing. */
  printf("\nRUNNING: motion %s %s %d %d %d %d",
         imageAFileName, imageBFileName, hMaxDisparity,
         vMaxDisparity, hConvNeighborhood, vConvNeighborhood);
  if (hRankNeighborhood > 0) {
```

```
    printf(" %d %d", hRankNeighborhood, vRankNeighborhood);
  }
  printf("\n\n");

#ifdef REPORT
  /* Print debugging info about the command line arguments. */
  printf("Max horizontal disparity: %d\nMax vertical disparity: %d\n",
         hMaxDisparity, vMaxDisparity);
  printf("Conv neighborhood width: %d\nConv neighborhood height: %d\n\n",
         hConvNeighborhood, vConvNeighborhood);
#endif

  /* Attempt to load in the input images, and check that attempt
     was successful. */
  imageA = (GrayImage)imLoad(IMAGE_GRAY, imageAFileName);
  imageB = (GrayImage)imLoad(IMAGE_GRAY, imageBFileName);
  if ((imageA == (GrayImage)NULL) || (imageB == (GrayImage)NULL)) {
    printf("Error loading one of the input images! Aborting.\n");
    exit(1);
  }

  /* Determine the dimensions of the input images, and do a sanity check. */
  imageWidth = (int)imGetWidth(imageA);
  imageHeight = (int)imGetHeight(imageA);

  if (((int)imGetWidth(imageB) != imageWidth) ||
      ((int)imGetHeight(imageB) != imageHeight)) {
    printf("Image dimensions do not match! Aborting.\n");
    exit(1);
  }

  /* The area of the images for which motion disparities can be computed
     is a function of the size of the match and search windows, and for
     rank transform, the rank neighborhood.  */
  hOffset = hMaxDisparity + hConvNeighborhood + hRankNeighborhood;
  vOffset = vMaxDisparity + vConvNeighborhood + vRankNeighborhood;

  croppedWidth = imageWidth - 2*(hOffset);
  croppedHeight = imageHeight - 2*(vOffset);

#ifdef REPORT
  /* Print debugging info about image dimensions. */
  printf("%ux%u input images loaded.\n", imageWidth, imageHeight);
  printf("Computable area = %ux%u\n", croppedWidth, croppedHeight);
  /*  printf("hOffset = %u\nvOffset = %d\n", hOffset, vOffset); */
#endif

  /* Create empty dx and dy motion disparity output images, checking
     for success. */
  dxDisparityImage =
    (ShortImage)imNew(IMAGE_SHORT, croppedWidth, croppedHeight);
  dyDisparityImage =
    (ShortImage)imNew(IMAGE_SHORT, croppedWidth, croppedHeight);

  imSetOffset(dxDisparityImage, hOffset, vOffset);
  imSetOffset(dyDisparityImage, hOffset, vOffset);

  if ((dxDisparityImage == (ShortImage)NULL) ||
      (dyDisparityImage == (ShortImage)NULL)) {
    printf("Problems creating output disparity images! Aborting.\n");
    exit(1);
  }

#ifdef REPORT
  printf("Motion disparity images created.\n\n");
#endif

  if (hRankNeighborhood > 0) {
    rankTransform(imageWidth, imageHeight);
  }
  else {
    normCorrelation(imageWidth, imageHeight);
  }

  /* Write motion disparity images to disk. */

  filename = (char *)malloc(25*sizeof(char));
  sprintf(filename, "dx_disparity.psm");

  writeDisparityImageFile(dxDisparityImage, filename,
                          hMaxDisparity, vMaxDisparity,
                          hConvNeighborhood, vConvNeighborhood);
#ifdef REPORT
  printf("Wrote %s.\n", filename);
#endif

  sprintf(filename, "dy_disparity.psm");

  writeDisparityImageFile(dyDisparityImage, filename,
                          hMaxDisparity, vMaxDisparity,
                          hConvNeighborhood, vConvNeighborhood);
```

```
#ifdef REPORT
  printf("Wrote %s.\n\n", filename);
#endif
  printf("background = %d", imRef(dxDisparityImage, 20, 20));
  /* Clean up the images that we used. */
  imFree(imageA);
  printf("Removed imageA.\n");
  imFree(imageB);
  printf("Removed imageB.\n");
  imFree(dxDisparityImage);
  printf("Removed dxDisparityImage.\n");
  imFree(dyDisparityImage);
  printf("Removed dyDisparityImage.\n");

  /* Get outta here. */
  return(0);
}

/* normCorrelation
 *
 *     input: integer dimensions of the input images
 *     output: void
 *
 *    Computes horizontal and vertical motion disparity images using
 *    normalized correlation.  Function is executed for effect on global
 *    output images.
 **/

void normCorrelation(int imageWidth, int imageHeight) {

  /* Declaration of variables local to _normCorrelation_. */
  FloatImage meanMatrix, stdDevMatrix;

  int xdiff, ydiff;
  int dataMatrixWidth, dataMatrixHeight;
  int croppedImageTop, croppedImageBottom, croppedImageLeft, croppedImageRight;
  int matchWindowTop, matchWindowBottom, matchWindowLeft, matchWindowRight;
  int matchWindowWidth, matchWindowHeight, N;
  int searchWindowTop, searchWindowBottom, searchWindowLeft, searchWindowRight;
  int imageI, imageJ;
  int searchI, searchJ;
  int matchI, matchJ;
  short dx, dy;
  float imageAMean, imageAStdDev, imageBMean, imageBStdDev;
  float minR, currentR;
  float runningSum;

#ifdef REPORT
  printf("Entering normalized correlation algorithm...\n");
#endif


  /* Figure out the boundaries of the "cropped image," the area of the
     image for which motion disparities are computable. */
  croppedImageTop = (vMaxDisparity + vConvNeighborhood);
  croppedImageBottom = (imageHeight - (vMaxDisparity + vConvNeighborhood) - 1);
  croppedImageLeft = (hMaxDisparity + hConvNeighborhood);
  croppedImageRight = (imageWidth - (hMaxDisparity + hConvNeighborhood) - 1);

  /* Create matrix images to store match window means and standard
     deviations as they are dynamically computed. Then set the origin of
     the matrices so that pixels match with the input images. Finish up
     with a sanity check. */
  dataMatrixWidth = (imageWidth - 2*hConvNeighborhood);
  dataMatrixHeight = (imageHeight - 2*vConvNeighborhood);

  meanMatrix =
    (FloatImage)imNew(IMAGE_FLOAT, dataMatrixWidth, dataMatrixHeight);
  stdDevMatrix =
    (FloatImage)imNew(IMAGE_FLOAT, dataMatrixWidth, dataMatrixHeight);

  imSetOffset(meanMatrix, hConvNeighborhood, vConvNeighborhood);
  imSetOffset(stdDevMatrix, hConvNeighborhood, vConvNeighborhood);

  if ((meanMatrix == (FloatImage)NULL) ||
      (stdDevMatrix == (FloatImage)NULL)) {
    printf("Could not create either meanMatrix or stdDevMatrix! Aborting.\n");
    exit(1);
  }


  /* Compute the dimensions of the match window, and the number of
     pixels in the window. */
  matchWindowWidth = (2*hConvNeighborhood + 1);
  matchWindowHeight = (2*vConvNeighborhood + 1);
  N = matchWindowWidth * matchWindowHeight;

  /* Loop over the area of the image for which motion disparities are
     computable. */
  for (imageJ = croppedImageTop; imageJ <= croppedImageBottom; imageJ++) {
```

```
#ifdef REPORT
    printf("Evaluating disparities for pixels in row %d.\n", imageJ);
#endif

    for (imageI = croppedImageLeft; imageI <= croppedImageRight; imageI++) {


      /* Initally, the minimum r is "infinity" and disparities are zero. */
      minR = 0;
      dx = 0;
      dy = 0;

      /* Find the boundaries of the match window for pixel (imageI, imageJ). */
      matchWindowTop = (imageJ - vConvNeighborhood);
      matchWindowBottom = (imageJ + vConvNeighborhood);
      matchWindowLeft = (imageI - hConvNeighborhood);
      matchWindowRight = (imageI + hConvNeighborhood);

      /* Compute mean and std dev for match window in image A. */
      imageAMean =
        bruteCalcMatchWindowMean(imageA, matchWindowTop, matchWindowBottom,
                                 matchWindowLeft, matchWindowRight);
      imageAStdDev =
        bruteCalcMatchWindowStdDev(imageA, imageAMean,
                                   matchWindowTop, matchWindowBottom,
                                   matchWindowLeft, matchWindowRight);

      /* Find the boundaries of the search window for pixel
         (imageI, imageJ). */
      searchWindowTop = (imageJ - vMaxDisparity);
      searchWindowBottom = (imageJ + vMaxDisparity);
      searchWindowLeft = (imageI - hMaxDisparity);
      searchWindowRight = (imageI + hMaxDisparity);


      /* Iterate over these search window in image B. */
      for (searchJ=searchWindowTop;searchJ<= searchWindowBottom; searchJ++) {
        for (searchI=searchWindowLeft;searchI<= searchWindowRight; searchI++) {

          /* Find the boundaries of the match window for
             the current location in the search window. */
          matchWindowTop = (searchJ - vConvNeighborhood);
          matchWindowBottom = (searchJ + vConvNeighborhood);
          matchWindowLeft = (searchI - hConvNeighborhood);
          matchWindowRight = (searchI + hConvNeighborhood);

          /* If we've been to this pixel before, we can save by grabbing
             the mean and std dev that we stored, otherwise we've got
             to compute them by brute force. */
          imageBMean = MEAN(searchI, searchJ);
          imageBStdDev = STD_DEV(searchI, searchJ);
          if ((imageBMean == 0) && (imageBStdDev == 0)) {
            /* Compute the mean and std dev, and save them for later. */
            imageBMean =
              bruteCalcMatchWindowMean(imageB,
                                       matchWindowTop, matchWindowBottom,
                                       matchWindowLeft, matchWindowRight);
            imageBStdDev =
              bruteCalcMatchWindowStdDev(imageB, imageBMean,
                                         matchWindowTop, matchWindowBottom,
                                         matchWindowLeft, matchWindowRight);
            STORE_MEAN(searchI, searchJ, imageBMean);
            STORE_DEV(searchI, searchJ, imageBStdDev);
          }

          /* Compute Pearson's r to determine to corrolation between the
             match window in image A and the current match window in the
             search window in image B. */

          runningSum = 0;
          for (matchJ=matchWindowTop;matchJ<=matchWindowBottom; matchJ++) {
            for (matchI=matchWindowLeft;matchI<=matchWindowRight; matchI++) {

              xdiff = matchI - searchI;
              ydiff = matchJ - searchJ;


              runningSum += ((I(imageA, (imageI + xdiff), (imageJ + ydiff))
                              - imageAMean) *
                             (I(imageB, matchI, matchJ) - imageBMean));
            }
          }
          currentR = (runningSum / ((N - 1) * imageAStdDev * imageBStdDev));

          /* If this location is the best yet, save it's disparity. */
          if ((1 - currentR) < (1 - minR)) {
            minR = currentR;
            dx = (searchI - imageI);
            dy = (searchJ - imageJ);
          }
        } /* End iteration over search window column. */
```

```
      } /* End iteration over search window row. */

      /* We've determined the dx and dy disparities that result in the
         minimum values for Pearson's r.  Now we should save them. */
      imRef(dxDisparityImage, imageI, imageJ) = (short)dx;
      imRef(dyDisparityImage, imageI, imageJ) = (short)dy;
    } /* Close loop over column in image. */
  } /* Close loop over row in image. */

#ifdef REPORT
  printf("Completed computation of disparities.  Exiting normCorrelation.\n");
#endif

  /* Clean up the images that we used. */
  imFree(meanMatrix);
  imFree(stdDevMatrix);
} /* Close _normCorrelation_ */


/* bruteCalcMatchWindowMean
 *
 *    input: image (either A or B) and boundaries of match window
 *    output: floating point mean of the intensities in the match window
 */
float bruteCalcMatchWindowMean(GrayImage image,
                               int matchWindowTop, int matchWindowBottom,
                               int matchWindowLeft, int matchWindowRight) {
  /* Declare variables local to _bruteCalcMatchWindowMean_. */
  int matchJ, matchI, N;
  float matchWindowMean, matchWindowSum;

  /* Compute number of pixels in match window. */
  N = ((matchWindowRight - matchWindowLeft + 1) *
       (matchWindowBottom - matchWindowTop + 1));
  matchWindowSum = 0;
  for (matchJ = matchWindowTop; matchJ <= matchWindowBottom; matchJ++) {
    for (matchI = matchWindowLeft; matchI <= matchWindowRight; matchI++) {
      matchWindowSum += I(image, matchI, matchJ);
    }
  }
  matchWindowMean = (matchWindowSum / N);
  return(matchWindowMean);
}

/* bruteCalcMatchWindowStdDev
 *
 *    input: image (either A or B) and boundaries of match window
 *    output: floating point standard deviation of the intensities in the
 *            match window
 */
float bruteCalcMatchWindowStdDev(GrayImage image,
                                 float matchWindowMean,
                                 int matchWindowTop, int matchWindowBottom,
                                 int matchWindowLeft, int matchWindowRight) {
  /* Declare variables local to _bruteCalcMatchWindowStdDev_. */
  int matchJ, matchI, N;
  float matchWindowStdDev, matchWindowSum;

  N = ((matchWindowRight - matchWindowLeft + 1) *
       (matchWindowBottom - matchWindowTop + 1));

  matchWindowSum = 0;
  for (matchJ = matchWindowTop; matchJ <= matchWindowBottom; matchJ++) {
    for (matchI = matchWindowLeft; matchI <= matchWindowRight; matchI++) {
      matchWindowSum += SQR(I(image, matchI, matchJ) - matchWindowMean);
    }
  }
  matchWindowStdDev = sqrt(matchWindowSum/(N - 1));

  return(matchWindowStdDev);
}

/* writeDisparityImageFile
 *
 * input: an image, filename, and information about the parameters that
 *        created the disparity image
 * output: void
 *
 * Writes the image to the file named by the given filename.
 */

void writeDisparityImageFile(ShortImage image, char *filename,
                    int hDisparity, int vDisparity,
                    int hNeighborhood, int vNeighborhood) {
  char *commentString;

  commentString = (char *)malloc(2000*sizeof(char));
  sprintf(commentString, "hDisp = %s\nvDisp = %s\nhNeigh = %s\nvNeigh = %s",
          hDisparity, vDisparity, hNeighborhood, vNeighborhood);
  imSetComment(image, commentString);
```

```
  if (imSave(image, filename) == -1) {
    printf("Error writing image file! Aborting.\n");
    exit(1);
  }
}

/* rankTransform
 *
 *     input: dimensions of the input images
 *     output: void
 *
 *    Computes horizontal and vertical motion disparity images using
 *    normalized correlation.  Function is executed for effect on global
 *    output images.
 **/
void rankTransform(int imageWidth, int imageHeight) {
  /* Declaration of variables local to _rankTransform_. */
  GrayImage rankTransformedImageA, rankTransformedImageB;
  ShortImage absDiffMatrix;
  int imageI, imageJ, searchI, searchJ, matchI, matchJ;
  int hOffset, vOffset;
  int croppedHeight, croppedWidth;
  int rankHeight, rankWidth;
  int croppedImageTop, croppedImageBottom, croppedImageLeft, croppedImageRight;
  int matchWindowTop, matchWindowBottom, matchWindowLeft, matchWindowRight;
  int searchWindowTop, searchWindowBottom, searchWindowLeft, searchWindowRight;
  short dx, dy, runningSum;
  int xdiff, ydiff;
  int minL;

  /* Figure out the boundaries of the "cropped image," the area of the
     image for which motion disparities are computable. */
  hOffset = (hMaxDisparity + hConvNeighborhood + hRankNeighborhood);
  vOffset = (vMaxDisparity + vConvNeighborhood + vRankNeighborhood);
  croppedHeight = (imageHeight - (2 * hOffset));
  croppedWidth = (imageWidth - (2 * vOffset));
  croppedImageTop = vOffset;
  croppedImageBottom = (vOffset + croppedHeight - 1);
  croppedImageLeft = hOffset;
  croppedImageRight = (hOffset + croppedWidth - 1);

  /* Create matix images to store pixel ranks as they are dynamically computed.
     Set the origin of the image to correspond to the cropped area of the
     image. Do a sanity check. */
  rankHeight = (imageHeight - (2 * vRankNeighborhood));
  rankWidth = (imageWidth - (2 * hRankNeighborhood));

  rankTransformedImageA =
    (GrayImage)imNew(IMAGE_GRAY, rankWidth, rankHeight);
  rankTransformedImageB =
    (GrayImage)imNew(IMAGE_GRAY, rankWidth, rankHeight);

  imSetOffset(rankTransformedImageA, hRankNeighborhood, vRankNeighborhood);
  imSetOffset(rankTransformedImageB, hRankNeighborhood, vRankNeighborhood);

  if ((rankTransformedImageA == (GrayImage)NULL) ||
      (rankTransformedImageB == (GrayImage)NULL)) {
    printf("Could not create rankTransformedImage! Aborting.\n");
    exit(1);
  }

  /* Perform rank transformations. */
  rankTransformedImageA = createRankTransformedImage(imageA, imageWidth,
                                                     imageHeight);
  rankTransformedImageB = createRankTransformedImage(imageB, imageWidth,
                                                     imageHeight);

  /* Create a matrix to store absolute difference values for the search
     window. */
  absDiffMatrix = (ShortImage)imNew(IMAGE_SHORT,
                                    (2 * hMaxDisparity + 1),
                                    (2 * vMaxDisparity + 1));

  if (absDiffMatrix == (ShortImage)NULL) {
    printf("Could not create absDiffMatrix! Aborting.");
    exit(1);
  }

  for (imageJ = vOffset; imageJ < (imageHeight - vOffset); imageJ++) {
#ifdef REPORT
    printf("Evaluating disparities for pixels in row %d.\n", imageJ);
#endif
    for (imageI = hOffset; imageI < (imageWidth - hOffset); imageI++) {

      /* Initialize minimum L_1 value and dx, dy disparities. */
      minL = INT_MAX;
      dx = 0;
      dy = 0;

      /* Find the boundaries of the match window in image A. */
      matchWindowTop = (imageJ - vConvNeighborhood);
```

```
      matchWindowBottom = (imageJ + vConvNeighborhood);
      matchWindowLeft = (imageI - hConvNeighborhood);
      matchWindowRight = (imageI + hConvNeighborhood);

      /* Find the boundaries of the search window for pixel
         (imageI, imageJ) and setup the matrix. */
      searchWindowTop = (imageJ - vMaxDisparity);
      searchWindowBottom = (imageJ + vMaxDisparity);
      searchWindowLeft = (imageI - hMaxDisparity);
      searchWindowRight = (imageI + hMaxDisparity);

      imSetOffset(absDiffMatrix, searchWindowLeft, searchWindowTop);
      /* Compute the sum of absolute differences for the UL pixel by
         brute force. */
      searchI = searchWindowLeft;
      searchJ = searchWindowTop;

      /* Iterate over the search window in image B. */
      for (searchJ=searchWindowTop;searchJ<= searchWindowBottom; searchJ++) {
        for (searchI=searchWindowLeft;searchI<= searchWindowRight; searchI++) {
          /* Find the boundaries of the match window in image B for
             the current location in the search window. */
          matchWindowTop = (searchJ - vConvNeighborhood);
          matchWindowBottom = (searchJ + vConvNeighborhood);
          matchWindowLeft = (searchI - hConvNeighborhood);
          matchWindowRight = (searchI + hConvNeighborhood);
          /* Compute pixels in top row of search window by brute force. */
          if (searchJ == searchWindowTop) {
            /* Compute the pixels in the top row by brute force. */
            runningSum = 0;
            for (matchJ=matchWindowTop;matchJ<=matchWindowBottom; matchJ++) {
              for (matchI=matchWindowLeft;matchI<matchWindowRight; matchI++) {
                xdiff = matchI - searchI;
                ydiff = matchJ - searchJ;

                /* Compute the sum of absolute differences. */
                runningSum += abs(I(rankTransformedImageA,
                                    (imageI + xdiff), (imageJ + ydiff)) -
                                  I(rankTransformedImageB, matchI, matchJ));
              }
            }
          }

          /* Use neighbor's absolute difference to make computation
             easier for all the rest of the pixels in the image. */
          else {
            /* Get the sum of absolute differences value for the match
               window neighboring one pixel above. */
            runningSum = imRef(absDiffMatrix, searchI, searchJ - 1);
            /* Remove the absolute differences for that row. */
            matchJ = (matchWindowTop - 1);

            for (matchI = matchWindowLeft;matchI<=matchWindowRight; matchI++) {
              xdiff = matchI - searchI;
              ydiff = matchJ - searchJ + 1;

              runningSum -= abs(I(rankTransformedImageA,
                                  (imageI + xdiff), (imageJ + ydiff)) -
                                I(rankTransformedImageB, matchI, matchJ));
            }
            /* Add the absolute differences for the row that's newly added
               to the match window. */
            matchJ = matchWindowBottom;
            for (matchI = matchWindowLeft;matchI<=matchWindowRight; matchI++) {
              xdiff = matchI - searchI;
              ydiff = matchJ - searchJ + 1;

              runningSum += abs(I(rankTransformedImageA,
                                  (imageI + xdiff), (imageJ + ydiff)) -
                                I(rankTransformedImageB, matchI, matchJ));
            }
          }
          /* If it's the best we've seen, save the disparity. */
          if (runningSum < minL) {
            minL = runningSum;
            dx = (searchI - imageI);
            dy = (searchJ - imageJ);
          }
          /* Store this absolute difference to make computation easier
             for neighboring pixels. */
          imRef(absDiffMatrix, searchI, searchJ) = runningSum;
        } /* Close loop over column in search window. */
      } /* Close loop over row in search window. */

      /* We've determined the dx and dy disparities that result in the
         minimum values for Pearson's r.  Now we should save them. */
      imRef(dxDisparityImage, imageI, imageJ) = (short)dx;
      imRef(dyDisparityImage, imageI, imageJ) = (short)dy;
    } /* Close loop over column in image. */
  } /* Close loop over row in image. */
```

```
#ifdef REPORT
  printf("Completed computation of disparities.  Exiting _rankTransform_.\n");
#endif

  /* Clean up the images that we used. */
  imFree(rankTransformedImageA);
  imFree(rankTransformedImageB);
  imFree(absDiffMatrix);
} /* Close _rankTransform_ */


/* createRankTransformedImage
 *
 *    input: input image and its dimensions
 *    ouput: rank transformed image
 *
 */
GrayImage createRankTransformedImage(GrayImage image, int imageWidth,
                                     int imageHeight) {

  /* Declare variables that are local to _createrankTransformedImage_. */
  GrayImage transformedImage;
  int croppedWidth, croppedHeight;
  int hOffset, vOffset;
  int rankWindowLeft, rankWindowRight, rankWindowTop, rankWindowBottom;
  int imageI, imageJ, rankI, rankJ;
  int rankCounter;
  float rankPixelIntensity, neighborsPixelIntensity;
  char *filename;

  /* We can only compute the rank transformed image for a cropped area,
     to ensure that the rank neighborhood is completely within the image. */
  hOffset = hRankNeighborhood;
  vOffset = vRankNeighborhood;
  croppedWidth = imageWidth - (2 * hOffset);
  croppedHeight = imageHeight - (2 * vOffset);

  /* Create an empty image, sets its origin to the proper offset, and do a
     sanity check. */
  transformedImage =
    (GrayImage)imNew(IMAGE_GRAY, croppedWidth, croppedHeight);
  imSetOffset(transformedImage, hOffset, vOffset);
  if (transformedImage == (GrayImage)NULL) {
    printf("Could not create transformedImage! Aborting.\n");
    exit(1);
  }

  /* Iterate over all the computable pixels in the image. */

  for (imageJ = (vOffset + 1); imageJ < (vOffset + croppedHeight); imageJ++) {
    for (imageI = hOffset; imageI < (hOffset + croppedWidth); imageI++) {
      rankWindowLeft = (imageI - hRankNeighborhood);
      rankWindowRight = (imageI + hRankNeighborhood);
      rankWindowTop = (imageJ - vRankNeighborhood);
      rankWindowBottom = (imageJ + vRankNeighborhood);
      rankPixelIntensity = I(image, imageI, imageJ);
      rankCounter = 0;

      /* For all the pixels in the rank neighborhood, count the number
         of pixels with leser intensity. */
      for (rankJ = rankWindowTop; rankJ <= rankWindowBottom; rankJ++) {
        for (rankI = rankWindowLeft; rankI <= rankWindowRight; rankI++) {
          if (I(image, rankI, rankJ) < rankPixelIntensity) {
            rankCounter++;
          }
        }
      }
      /* Store the rank for this pixel. */
      imRef(transformedImage, imageI, imageJ) = rankCounter;
    } /* Close loop over rows in the image. */
  } /* Close loop over columns in the image. */

  /* We're done, get outta here. */
  return(transformedImage);
}
```