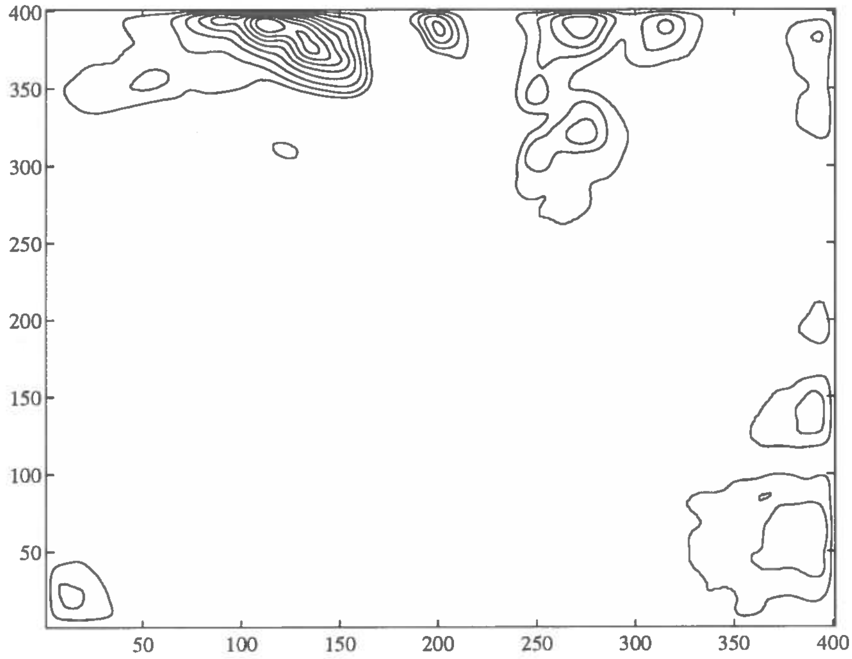

Figure B.6: **NG40** *Contour Map (Smoothed* ×40*).*

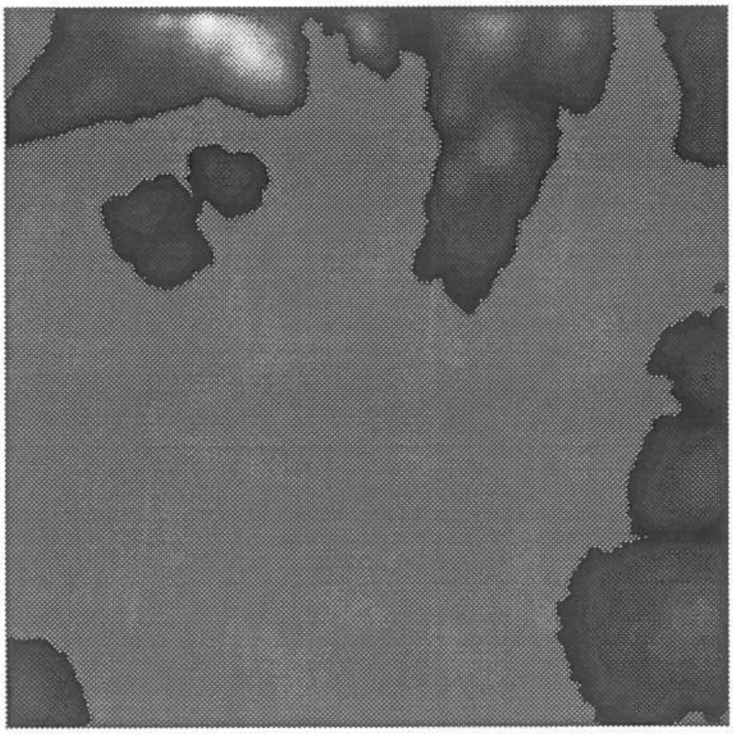

Figure B.7: **NG40** *Range Image (Smoothed* ×40*).*

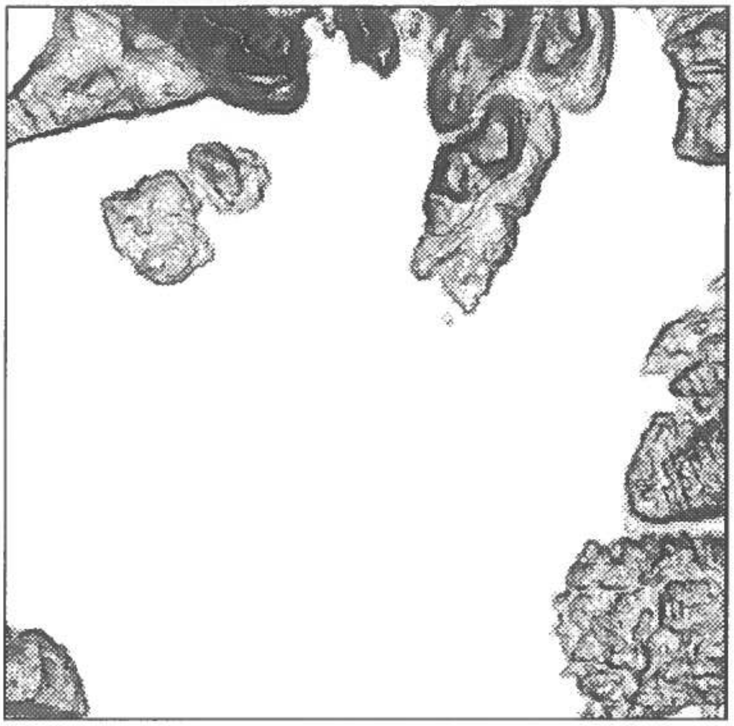

Figure B.8: **NG40** *Cosine Shaded Image.*

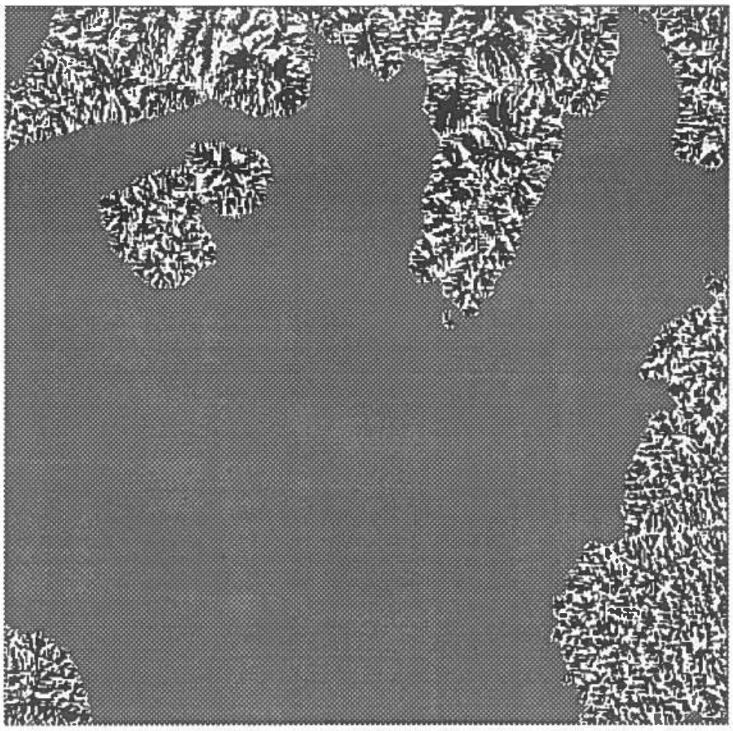

Figure B.9: **NG40** *Classified Image (Original).*
white ⇒ valley; black ⇒ ridge; grey ⇒ flat.

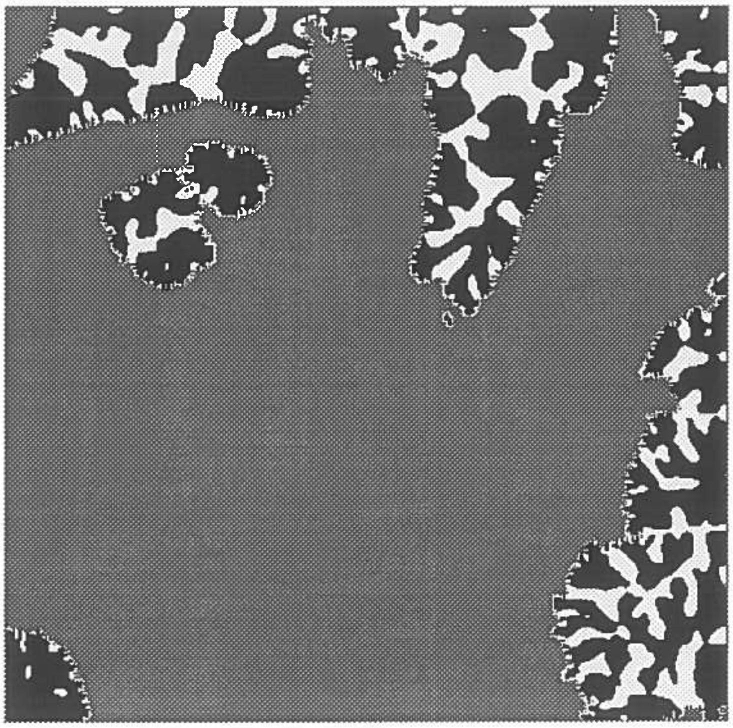

Figure B.10: **NG40** *Classified Image (Smoothed* ×40*).*

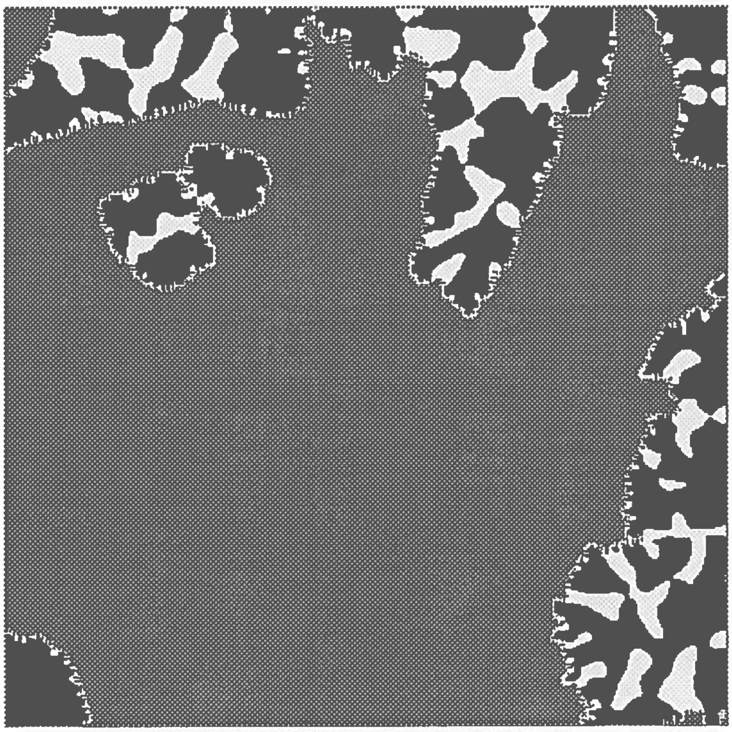

Figure B.11: **NG40** *Classified Image (Smoothed* ×80*).*
white ⇒ valley; black ⇒ ridge; grey ⇒ flat.

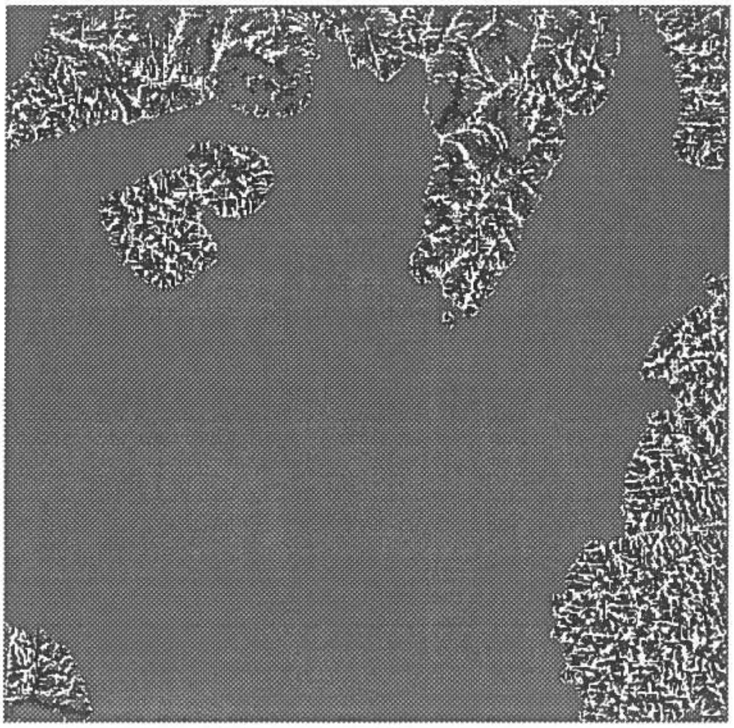

Figure B.12: *Thresholded* **NG40** *Classified Image (Original).*

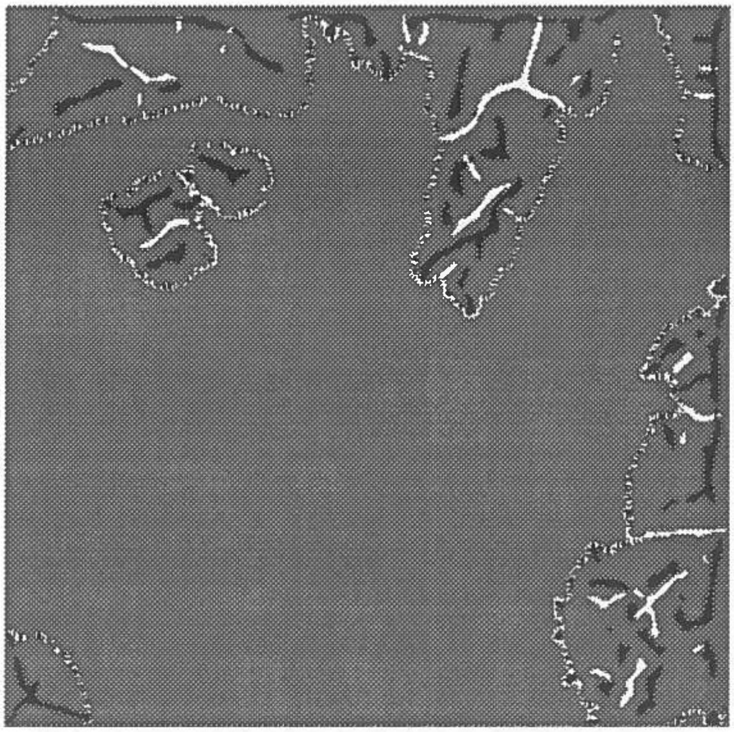

Figure B.13: *Thresholded* **NG40** *Classified Image (Smoothed* $\times 40$*).*
white $\Rightarrow$ *valley; black* $\Rightarrow$ *ridge; grey* $\Rightarrow$ *flat.*

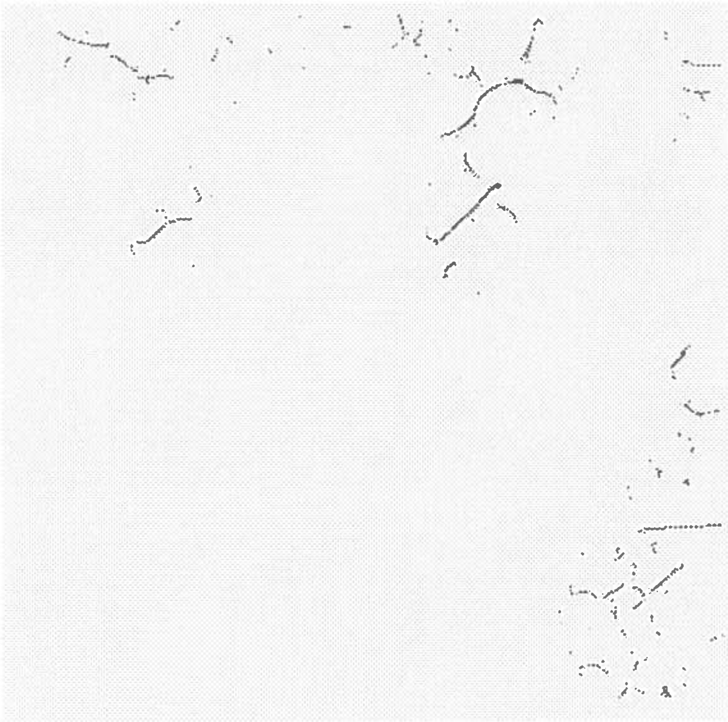

Figure B.14: **NG40** *(Smoothed ×40) Unconnected Valley Minima.*

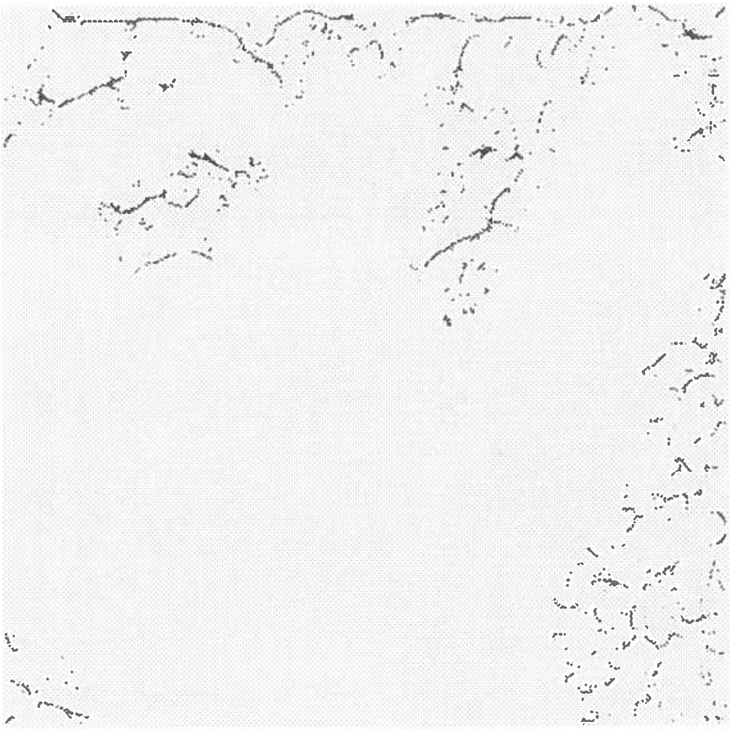

Figure B.15: **NG40** *(Smoothed ×40) Unconnected Ridge Apexes.*

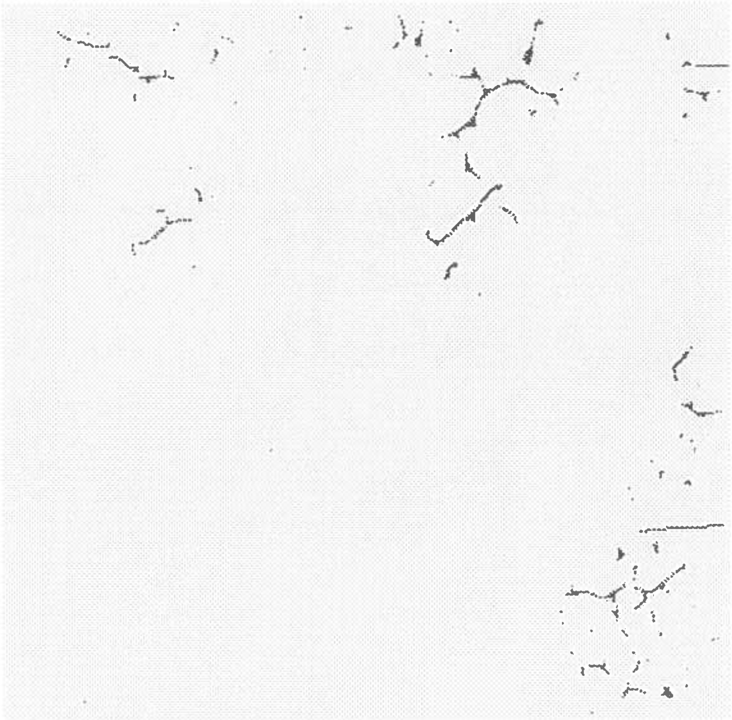

Figure B.16: *Connecting Gaps in the* **NG40** *Valley Minima Tracks.*

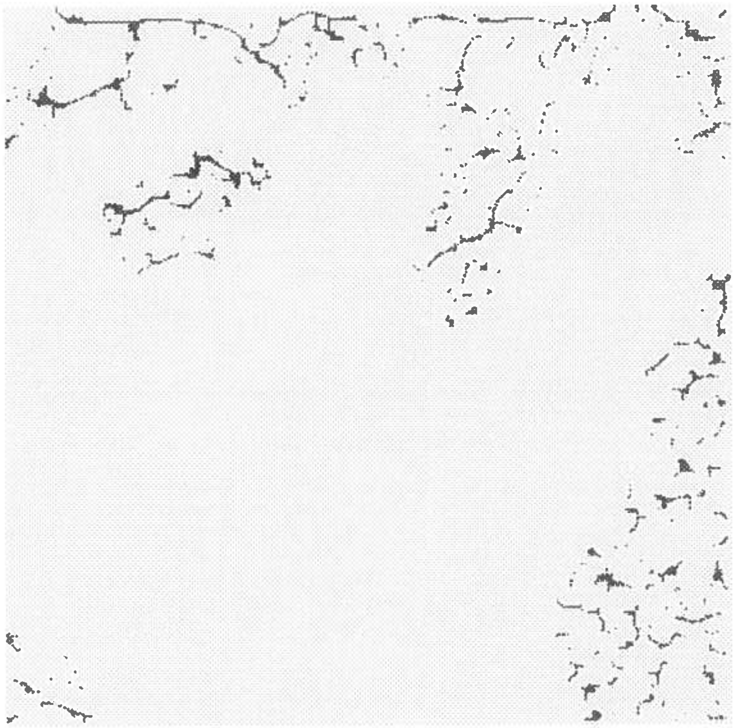

Figure B.17: *Connecting Gaps in the* **NG40** *Ridge Apex Tracks.*

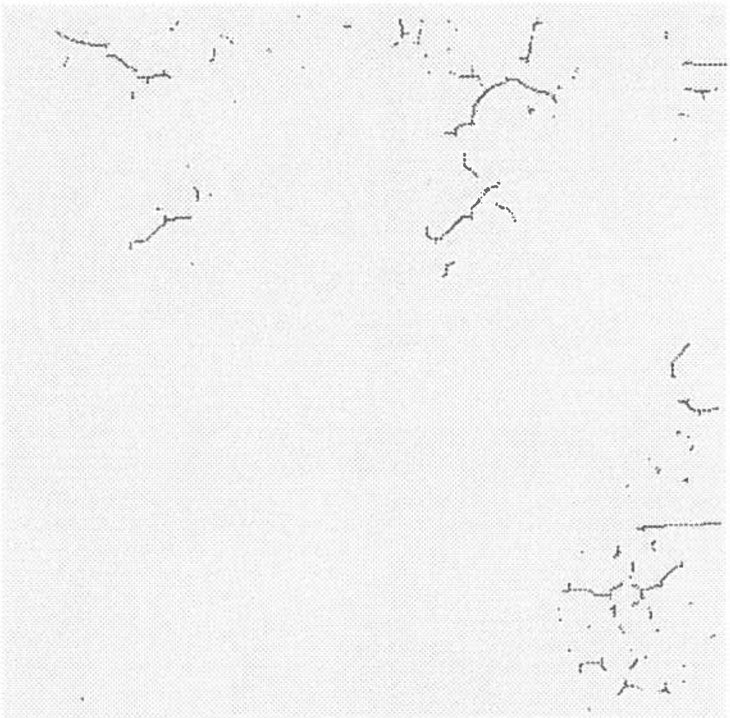

Figure B.18: *The Effect of Track Thinning (***NG40** *Valley Minima).*

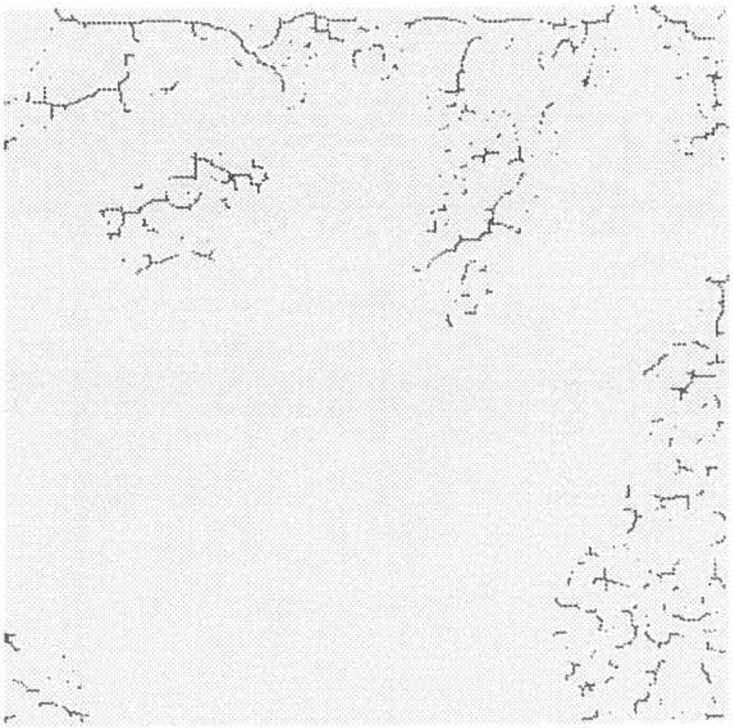

Figure B.19: *The Effect of Track Thinning (***NG40** *Ridge Apexes).*

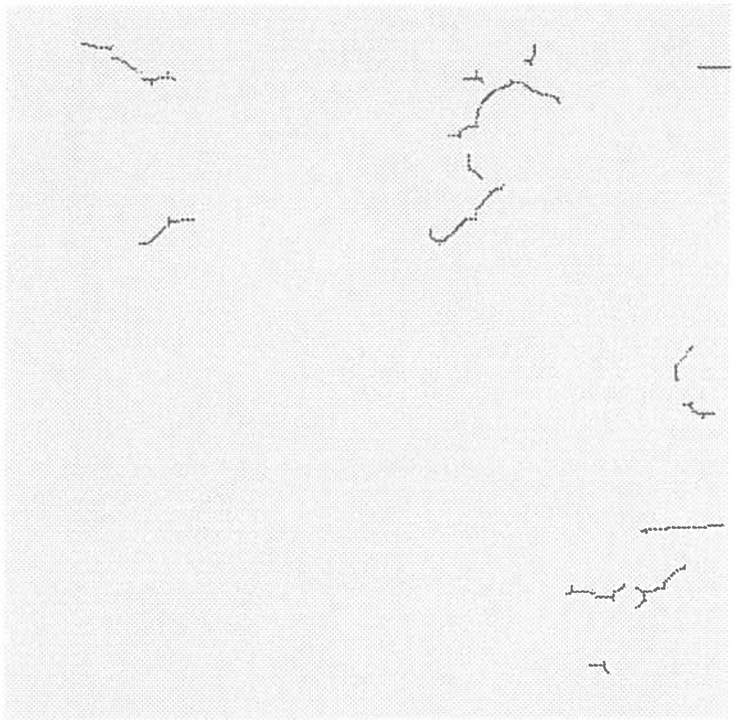

Figure B.20: **NG40** *Valley Minima of < 20 Pixels Removed.*

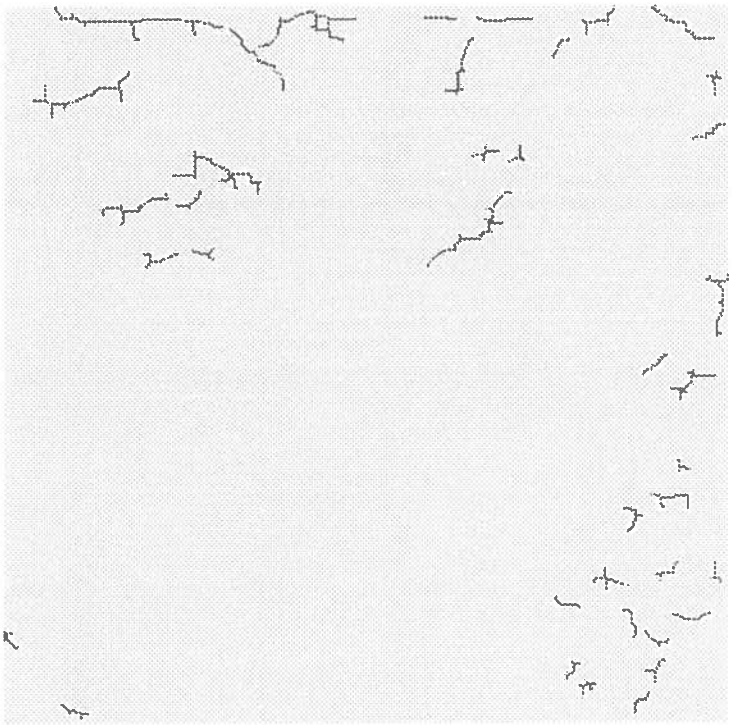

Figure B.21: **NG40** *Ridge Apexes of < 20 Pixels Removed.*

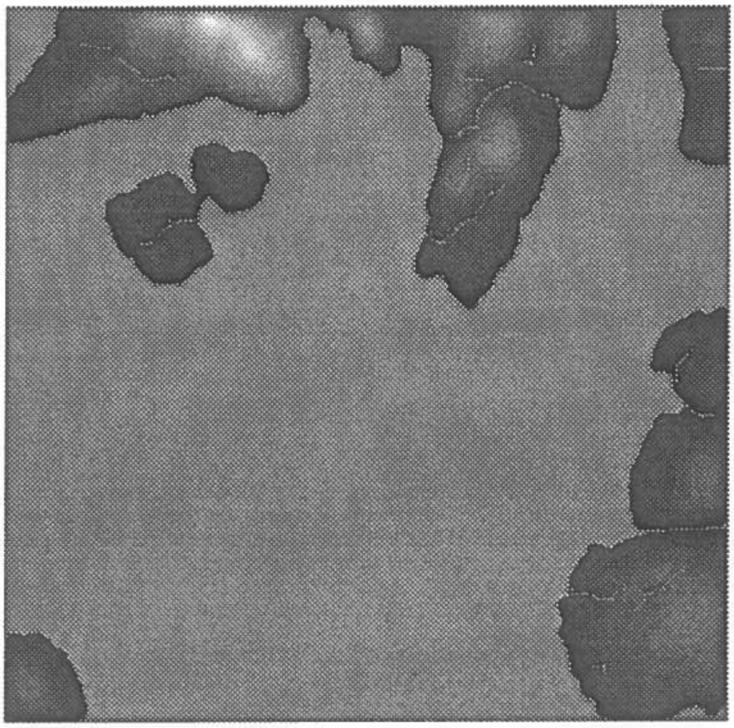

Figure B.22: **NG40** *Overlaid Valley Minima.*

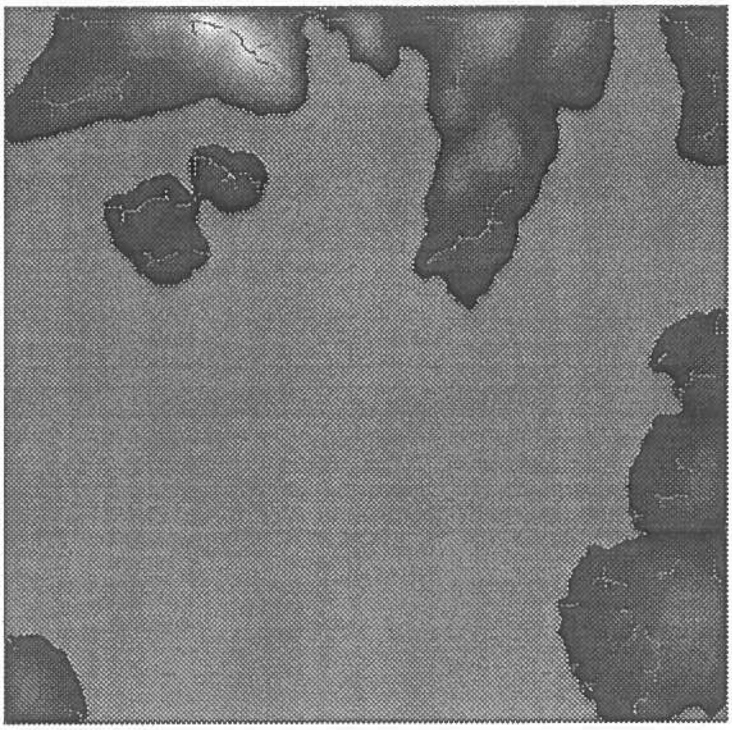

Figure B.23: **NG40** *Overlaid Ridge Apexes.*

Appendix C

Program Code

C.1 Data Smoothing & Preprocessing

C.1.1 preprocess.cxx

```
/*      Preprocesses an NTF data file into a suitable
 *      400x400 pixel array of values, which is
 *      smoothed via <int> iterations, as specified
 *      by the -s command line value.
 *
 *      usage : preprocess -i <infile> -o <outfile> -s int
 */

#include <stdio.h>

extern array2hips(float *, int, char [20]);

/*
 *      This funtion reads 4 ASCII characters from
 *      the file pointed to by fileptr, and
 *      returns the corresponding integer value.
 */
float ascii_height(FILE *fileptr)
{
        char *cptr = (char *) malloc (4 * sizeof(char));

        cptr[0] = getc(fileptr);
        cptr[1] = getc(fileptr);
        cptr[2] = getc(fileptr);
        cptr[3] = getc(fileptr);

        return ((float) atoi(cptr));
}


/*
 *      Function to smooth the original data using
 *      a 3x3 convolution operator mask. The mask
 *      is fitted around each pixel, with the
 *      centre pixel getting the sum of the neigh-
 *      bouring pixel products divided by 24.
 *
 *      1       2       1
 *      2       12      2
 *      1       2       1
 *
 *      The smoothing is done a number of times
 *      specified by 'iterations' which was given
 *      as a command line arguement.
 *
 *      Data coords range from 0:400 and 0:400
 *      although we loop from 1:399 and 1:399
 *      as the mask cannot be applied at the
 *      extreme edges of the image. The height
 *      values are in the range -100:1500m.
 */
void smooth(float **data, int iterations)
{
        int i, j, loop;
        float **newdata;

        /*      allocate memory for updated values array        */
        newdata = (float **) malloc (401 * sizeof (float *));
        newdata[0] = (float *) malloc (401 * 401 * sizeof (float));
        for (i = 1; i < 401; i++)
                newdata[i] = newdata[0] + i * 401;

        for (loop = 0; loop < iterations; loop++)
        {
                fprintf(stderr, "preprocess : smoothing iteration %d / %d\n",
```



```
                                                loop+1, iterations);
                for (i = 1; i < 400; i++)
                {
                        for (j = 1; j < 400; j++)
                        {
                                if (data[i][j] == 0) /* ignore sea-level pixels */
                                {
                                        newdata[i][j] = 0;
                                        continue;
                                }

                                newdata[i][j] = data[i-1][j-1] +
                                                data[i+1][j-1] +
                                                data[i-1][j+1] +
                                                data[i+1][j+1] +
                                                data[i][j-1] * 2 +
                                                data[i][j+1] * 2 +
                                                data[i-1][j] * 2 +
                                                data[i+1][j] * 2 +
                                                data[i][j] * 12;

                                newdata[i][j] /= 24;
                        }
                }

                for (i = 0; i < 401; i++)                       /* next iteration */
                        for(j = 0; j < 401; j++)                /* use new values */
                                data[i][j] = newdata[i][j];
        }
}


/*
 *      Main program begin.
 */
main(int argc, char *argv[])
{
        int     i, j, count, loop, iterations;
        FILE    *in_fileptr, *out_fileptr;

        char    *rubbish = (char *) malloc (80 * sizeof (char)),
                in_file[20],
                out_file[20];

        float   **data;

        if (argc != 7)                                  /* enough command line args */
        {
                fprintf(stderr, "usage : data2hips -i <file> -o <file> -s int\n");
                exit(0);
        }

        for (loop = 1; loop < argc; loop++)             /* check command line args */
        {
            if (argv[loop][0] == '-')
            {
                switch (argv[loop][1])
                {
                case 'o' :                              /* output filename */
                        sscanf(argv[++loop], "%s", out_file);
                        break;
                case 'i' :                              /* input filename */
                        sscanf(argv[++loop], "%s", in_file);
                        break;
                case 's' :                              /* input filename */
                        sscanf(argv[++loop], "%d", &iterations);
```

```
                                break;
                        default :                       /* error arg */
                                printf(stderr, "usage : data2hips -i <file> -o <file> -s int\
n");
                                exit(0);
                        }
                }
        }

        if ((in_fileptr = fopen(in_file, "r")) == NULL)
        {
                fprintf(stderr, "data2hips : can't open input file : %s\n", in_file);
                exit(0);
        }

        /*      allocate memory dynamically     */

        data = (float **) malloc (401 * sizeof (float *));
        data[0] = (float *) malloc (401 * 401 * sizeof (float));
        for (i = 1; i < 401; i++)
                data[i] = data[0] + i * 401;

    for (loop = 0; loop < 401; loop++)          /* # of data blocks */
    {
        fprintf(stderr, "preprocess : reading block[%d]\n", loop);

        count = 400;
        fgets(rubbish, 80, in_fileptr);         /* ignore '51 xxx' line */
        for (j = 0; j < 21; j++)                /* 21 lines per block */
        {
                for (i = 0; i < 19; i++)        /* 19 values per line */
                        data[count--][loop] = ascii_height(in_fileptr);

                fgets(rubbish, 10, in_fileptr); /* ignore ending '1' char */
        }

        for (i = 1; i >= 0; i--)                /* final two values */
                data[i][loop] = ascii_height(in_fileptr);

        fgets(rubbish, 10, in_fileptr);         /* ignore ending '0' char */
    }

        fclose(in_fileptr);

        if (iterations > 0) smooth(data, iterations);

        printf("preprocess : creating HIPS image\n");
        array2hips(*data, 401, out_file);       /* create HIPS image */
}
```

C.2 Local Curvature, Shape & Orientation Calculation

C.2.1 process.cxx

C.2.2 hkcode.cxx

C.2.3 orient.cxx

```
/*
 *      Process a HIPS image taken from stdin, by doing
 *      the following steps :
 *
 *      [1] : Uses a winsize * winsize window to look at
 *              each image pixel.
 *      [2] : Fits a biquadratic surface about each pixel
 *              (if possible), using data from the window.
 *      [3] : Calculates the mean curvature (H) value for
 *              each pixel using the biquadratic fit.
 *      [4] : Calculates the tan of the angle that the
 *              minimum curvature of each pixel makes
 *              for use during suppression.
 *      [5] : Dumps the array of H and tan values to
 *              separate files for use during suppression.
 *      [6] : Generates, and dumps to file, a cosine shaded
 *              image.
 *
 *      Usage : process -i file_id [-w integer] < HIPS_image
 *
 *      The -i flag specifies the identifier which will
 *      be used to tag all dump files and HIPS images
 *      that are generated along the way.
 *
 *      The -w option specifies the n x n size of the
 *      window which is used to fit the bidquadratic
 *      surface. The default size is 3 (the window is
 *      always square).
 */

#include <stdio.h>
#include <string.h>
#include <hipl_format.h>

#define FALSE 0
#define TRUE 1

char Progname[]="process";              // needed by <hipl_format.h>

extern float calc_H(double *par, float x, float y);
extern double tancalc(double *par, float x, float y, float *coshaded);
extern array2hips(float *data, int side, char file[20], float min, float max);

//      Andrew Fitzgibbon's library SVD Stuff

extern double **nr_matrix(int nrl,int nrh,int ncl,int nch);
extern double *nr_vector(int nl,int nh);
extern void nr_free_matrix(double **m,int nrl,int nrh,int ncl,int /* nch*/);
extern void nr_free_vector(double *v,int nl,int /*nh*/);
extern void build_svd_stuff(int offset, int npts, double ** U, double ** v, double *
w);
extern void svbksb(double ** u, double * w, double ** v, int m, int n, double * b, do
uble * x);


/*
 *      Function to apply window from top to
 *      bottom, left to right. Generates a
 *      list of height values at each of the
 *      [x,y] points within the window.
 *
 *      The function is passed the 'answer' into
 *      which to place the required results. This
 *      is allocated the required memory at
 *      the beginning of the main program, and
 *      freed accordingly at the end of processing.
```



```
 */
void apply_window(int left, int top, int right, int bottom, int cols,
                float *data, double *Zlist)
{
        int step = 1;
        for (int i = left; i <= right; i++)
                for (int j = top; j <= bottom; j++)
                        Zlist[step++] = data[i * cols + j];
}


/*
 *      Function to check that the window
 *      does not access the array of image
 *      pixels out of bounds, ie. values
 *      less than zero or > max which would
 *      cause an error. If part of window
 *      falls out of range then we cannot
 *      calculate the H or K value for
 *      that window's pixel.
 */
int window_in_range(int a, int b, int x, int y, int max)
{
        if ( (a < 0) || (b < 0) || (x > max) || (y > max) )

                return FALSE;           // out of range
                return TRUE;            // in range
}


/*
 *      Function to dump the arrays of H and tan
 *      values to separate files, named by the
 *      following convention, according to the
 *      user give file_id (specified through the
 *      -i command line argument) :
 *
 *      h_identifier    (for the H array)
 *      t_identifier    (for the tan array)
 */
void dump_to_file(int size, float *H, float *T, char file_id[20])
{
        FILE *H_fileptr, *T_fileptr;

        char    h[22] = "h_",
                t[22] = "t_";

        strcat(h, file_id);                     // generate the filenames
        strcat(t, file_id);

        if (((H_fileptr = fopen(h, "w")) == NULL)       // open files if poss
         || ((T_fileptr = fopen(t, "w")) == NULL))
        {
                fprintf(stderr, "process : can't open dump files\n");
                exit(0);
        }

        fprintf(stderr, "\n\n process  : dumping details to files\n");
        fprintf(H_fileptr, "%d\n", size);
        fprintf(T_fileptr, "%d\n", size);       // first dump array sizes

        for (int loop = 0; loop < size; loop++) // then all the values
        {
                fprintf(H_fileptr, "%g\n", H[loop]);
                fprintf(T_fileptr, "%g\n", T[loop]);
        }
```

```
        fclose(H_fileptr);                      // close files
        fclose(T_fileptr);
}


/*
 *      main program begin
 */
main(int argc, char *argv[])
{
        struct  header hd;
        char    file_id[20];                    // identifier for dump files
        float   shade;                          // pixel cosine shaded value
        int     curr_row = 500, loop,           // some stuff
                picsize,                        // no. of image pixels
                winsize = 3,                    // 3 => default
                x, y,                           // pixel coords
                offset,                         // window offset from [x,y]
                xbegin, xend,                   // window x-coord limits
                ybegin, yend,                   // window y-coord limits
                no_of_points;                   // in the window

        if (argc < 3)                           // enough command line args ?
        {
                fprintf(stderr, "usage : process -i file_id [-w integer] < HIPS_image
\n");
                exit(0);
        }

        for (loop = 1; loop < argc; loop++)     // check command line args
        {
            if (argv[loop][0] == '-')
            {
                switch (argv[loop][1])
                {
                case 'w' :                      // specify window size
                        sscanf(argv[++loop], "%d", &winsize);
                        break;
                case 'i' :                      // specify file identifier
                        sscanf(argv[++loop], "%s", file_id);
                        break;
                default :                       // sorry, illegal arg
                        fprintf(stderr, "usage : process -i file_id [-w integer] < HI
PS_image\n");
                        exit(0);
                        break;
                }
            }
        }

        read_header(&hd);                       // read image specs
        if (hd.pixel_format != PFFLOAT)         // pixels must be floats
        {
                fprintf(stderr, "process : incorrect image pixel format\n");
                exit(0);
        }

        offset = (winsize - 1) / 2;
        picsize = hd.rows * hd.cols;
        no_of_points = winsize * winsize;

        fprintf(stderr, "\n Window Size    : %d\n", no_of_points);
        fprintf(stderr, " Picture Size   : %d\n", picsize);

        double * biquad_params = new double [6];        // memory allocation
```



```
        float   * H = new float [picsize];
        float   * T = new float [picsize];
        float   * S = new float [picsize];
        float   * data = new float [picsize];

        pread(0, data, picsize*sizeof(float));          // read in HIPS data

        /*
         *      allocate space for matrices and
         *      vectors to be used in the SVD
         *      fit (Andrew FG's code) and then
         *      generate matrix U and vectors V, W
         *      needed for the biquadratic fit.
         */
        double ** U = nr_matrix(1, no_of_points, 1, 6);
        double ** V = nr_matrix(1, 6, 1, 6);
        double  * W = nr_vector(1, 6);
        double  * Z = nr_vector(1, no_of_points);
        build_svd_stuff(offset, no_of_points, U, V, W);

        //      process every image pixel

        for (loop = 0; loop < picsize; loop++)
        {
                y = loop % hd.cols;             // calc 2D coords from 1D data
                x = (loop - y) / hd.cols;

                if (curr_row != x) {            // per row progress report
                  curr_row = x;
                  if (loop % 10 == 0)
                    fprintf(stderr, "\n Processing Row : %3d", curr_row);
                  else fprintf(stderr, " %3d", curr_row);
                }

                //      calculate window limits and check
                xbegin = x - offset; xend = x + offset;
                ybegin = y - offset; yend = y + offset;

                if (!window_in_range(xbegin, ybegin, xend, yend, hd.cols - 1))
                {
                        H[loop] = T[loop] = -9;         // dummy values
                        continue;                       // next pixel in loop
                }
                else                                    // window in range
                {
                    // apply the window to get the list of height (z) values
                    apply_window(xbegin, ybegin, xend, yend, hd.cols, data, Z);

                    // generate SVD fit to calculate biquadratic fit parameters
                    svbksb(U, W, V, no_of_points, 6, Z, biquad_params-1);

                    // calculate mean curvature (H) value
                    // the tan of the major axis
                    // and the cosine shaded pixel value
                    H[loop] = calc_H(biquad_params, x, y);
                    T[loop] = (float) tancalc(biquad_params, x, y, &shade);
                    S[loop] = shade;
                }
        }

        //      generate and dump the cosine shaded HIPS image
        //      with the filename in accord with the identifier :
        //
        //      identifier_shaded       (the image filename)

        char result[30] = "";
```

```
        strcpy(result, file_id);
        strcat(result, "_shaded");
        array2hips(S, hd.cols, result, 0, 1);

        dump_to_file(picsize, H, T, file_id);   // dump the arrays to file

        delete H;                               // free allocated memory
        delete T;
        delete S;
        delete data;
        delete biquad_params;

        nr_free_matrix(U, 1, no_of_points, 1, 6);
        nr_free_matrix(V, 1, 6, 1, 6);
        nr_free_vector(W, 1, 6);
        nr_free_vector(Z, 1, no_of_points);

        fprintf(stderr, "\n");
}
```

```
/*
 *      This file contains functions to calculate the mean (H)
 *      and gaussian (K) curvatures at some given point on a
 *      biquadratic fit. The arguments required are :
 *
 *              (1) an array of the biquadratic parameters
 *              (2) x coord (usually zero due to transposing)
 *              (3) y coord (usually zero due to transposing)
 *
 *      To calculate the two principle curvatures, use:
 *
 *              P1 = H - sqrt(H*H-K)
 *              P2 = H + sqrt(H*H-K)
 */

#include <math.h>

float calc_K(double *par,float x, float y);
float calc_H(double *par,float x, float y);

float calc_K(double *par,float x, float y)
{
  float a,b,c,d,e,f;
  float zxx,zyy,zxy,zx,zy;      /* the various derivatives of z */
  float K;                      /* Gaussian curvature */

  a=par[0];                     /* This is just for easy reading */
  b=par[1];
  c=par[2];
  d=par[3];
  e=par[4];
  f=par[5];

  zxx=2*d;
  zyy=2*e;
  zxy=f;
  zx=b;                 // b + 2*d*x + f*y      (but x = 0 = y)
  zy=c;                 // c + 2*e*y + f*x      (but x = 0 = y)

  K = ( zxx*zyy-zxy*zxy ) / ( (1+zx*zx+zy*zy)*(1+zx*zx+zy*zy) );

  return K;

}

float calc_H(double *par,float x,float y)
{
  float a,b,c,d,e,f;
  float zxx,zyy,zxy,zx,zy;      /* the various derivatives of z */
  float H;                      /* Mean curvature */
  float top,bottom;

  a=par[0];                     /* This is just for easy reading */
  b=par[1];
  c=par[2];
  d=par[3];
  e=par[4];
  f=par[5];

  zxx=2*d;
  zyy=2*e;
  zxy=f;
  zx=b;                 // b + 2*d*x + f*y      (but x = 0 = y)
  zy=c;                 // c + 2*e*y + f*x      (but x = 0 = y)
```



```
  top = zxx+zyy+zxx*zy*zy+zyy*zx*zx-2*zx*zy*zxy;

  bottom =  2*(sqrt(1+zx*zx+zy*zy))*(sqrt(1+zx*zx+zy*zy))*(sqrt(1+zx*zx+zy*zy));

  H = top/bottom;

  return H;
}
```

```
/*
 *      This function calculates the tan of the
 *      angle that the major curvature axis
 *      makes with the x-axis, (towards the
 *      y-axis).
 *
 *      This is done by working out both the
 *      principal curvature magnitudes and
 *      choosing the smaller in order to work
 *      out the (u,v)-direction vectors of
 *      the minimum curvature.
 *
 *      The tan the minimum curvature axis makes
 *      is then just :
 *
 *              tan B = v / u
 *
 *      We require three arguments :
 *
 *      [1] : the array of biquadratic parameters.
 *      [2] : the x-coordinate.
 *      [3] : the y-coordinate.
 *
 *      The function also calculates the cosine
 *      shaded pixel value in order to build up
 *      an overall cosine shaded image, as it
 *      requires much of the same maths needed
 *      in calculating the curvatures etc.
 */

#include<math.h>

double tancalc(double *par, float x, float y, float *coshaded)
 {
        double  a, b, c, d, e, f,
                root, top,
                zx, zy, zxx, zyy, zxy;

        a = par[0]; b = par[1]; c = par[2];     /* easy reading */
        d = par[3]; e = par[4]; f = par[5];

        zx  = b;                // + 2*d*x + f*y; (if not transposed)
        zy  = c;                // + 2*e*y + f*x; (if not transposed)
        zxx = 2*d;
        zxy = f;
        zyy = 2*e;

        double E = 1 + zx * zx;
        double F = zx * zy;
        double G = 1 + zy * zy;

        //      generate the cosine shaded pixel value
        root = sqrt(1 + zx*zx + zy*zy);
        *coshaded = (float) 1 / root;

        double L = zxx / root;
        double M = zxy / root;
        double N = zyy / root;

        double S = E*G - F*F;

        double A = G*L - F*M;
        double B = G*M - F*N;
        double C = E*M - F*L;
        double D = E*N - F*M;
```



```
        root = sqrt( (D - A)*(D - A) + 4*B*C );

        double C1 = (D + A + root) / 2;         // one curvature
        double C2 = (D + A - root) / 2;         // the other curvature

        if (fabs(S*C1 - A) < 1e-10)             // avoids dividing into zero
          return 0;

        if (fabs(B) < 1e-10)                    // avoids divisons by zero
          return infinity();

        //      Compare magnitudes and return the tan
        //      the smallest one makes with the x-axis.
        //      This is the minimum curvature, ie. the
        //      curvature along the feature

        if (fabs(C1) < fabs(C2))
                return ((S*C1 - A) / B);
        else
                return ((S*C2 - A) / B);
 }
```

C.3 Non–Maximal Suppression & Tracking

C.3.1 suppress.cxx

C.3.2 remove.cxx

```
/*
 *      Performs a non-maximal suppression type
 *      algorithm to identify the candidate points
 *      which are the bottom of valleys and the
 *      tops of ridges.
 *
 *      Each pixel can be classified according to
 *      the sign of its mean curvature (H) value
 *      calculated during the image processing
 *      stage, according to the following :
 *
 *      H < 0   =>      ridge-type point
 *      H > 0   =>      valley-type pixel
 *      H = 0   =>      plane, minimal etc.
 *
 *      These bands can be augmented using a
 *      threshold value, to allow for almost
 *      zero values, and/or to enable us to
 *      ignore "weak" candidate pixels, ie :
 *
 *      H < +threshold  =>      ridge-type point
 *      H > -threshold  =>      valley-type pixel
 *
 *      All in all, I suppose its tracking by
 *      cheating - without using hysteresis,
 *      gradient descent, or anything of the like !
 *
 *      Usage : suppress -i file_id [-t threshold] < HIPS_image
 *
 *      The -i flag specifies the identifier which will
 *      be used to tag the two generated HIPS images of
 *      the "tracked" valleys and ridges.
 *
 *      The -t option allows the user to specify a
 *      threshold value to be used in tolerating
 *      +ve and -ve deviations from zero for the
 *      mean curvature (H) values. There is a
 *      built-in default of 0.0 (surprisingly !)
 */

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

#define deg2rad(x)  ((x * M_PI) / 180)    // a nice conversion function
#define rad2deg(x)  ((x * 180) / M_PI)    // another nice function

#define TRUE 1
#define FALSE 0

char Progname[]="suppress";             // needed by <hipl_format.h>

extern array2hips(float *, int, char [20], float min, float max);


/*
 *      Function returns TRUE iff value is strictly
 *      less than or equal to both alpha and beta.
 */
float minimum(float value, float alpha, float beta)
{
        if ((value <= alpha) && (value <= beta))
                return TRUE;
        else
```



```
                return FALSE;
}


/*
 *      Function returns TRUE iff value is strictly
 *      greater than or equal to both alpha and beta.
 */
float maximum(float value, float alpha, float beta)
{
        if ((value >= alpha) && (value >= beta))
                return TRUE;
        else
                return FALSE;
}


/*
 *      main program begin
 */
main(int argc, char *argv[])
{
        struct header hd;
        char file_id[20], thresh_id[8] = "0.0";
        FILE *t_fileptr, *h_fileptr;
        int t_size, h_size;
        double threshold = 0.0;                 // default value

        if (argc < 3)                           // enough command line args ?
        {
                fprintf(stderr, "usage : suppress -i file_id [-t threshold] < HIPS_im
age\n");
                exit(0);
        }

        for (int loop = 1; loop < argc; loop++) // check command line args
        {
            if (argv[loop][0] == '-')
            {
                switch (argv[loop][1])
                {
                case 't' :                      // specify threshold
                        sscanf(argv[++loop], "%s", thresh_id);
                        threshold = atof(thresh_id);
                        fprintf(stderr, "\n h_threshold = %f\n", threshold);
                        break;
                case 'i' :                      // specify file identifier
                        sscanf(argv[++loop], "%s", file_id);
                        break;
                default :                       // sorry, illegal arg
                        fprintf(stderr, "usage : suppress -i file_id [-t threshold] <
 HIPS_image\n");
                        exit(0);
                        break;
                }
            }
        }

        char    t_filename[22] = "t_",          // generate the filenames
                h_filename[22] = "h_";

        strcat(t_filename, file_id);
        strcat(h_filename, file_id);

        if (((t_fileptr = fopen(t_filename, "r")) == NULL)
         || ((h_fileptr = fopen(h_filename, "r")) == NULL))
```

```
        {
                fprintf(stderr, "suppress : can't open dump files\n");
                exit(0);
        }

        fscanf(t_fileptr, "%d\n", &t_size);     // read in array sizes
        fscanf(h_fileptr, "%d\n", &h_size);
        if (t_size != h_size)                   // which must be the same
                {
                        fprintf(stderr, "suppress : H & T file sizes incompatible\n")
;
                        exit(0);
                }

        //      Then lettuce begin !!

        read_header(&hd);
        if ((hd.pixel_format != PFFLOAT) || (t_size != (hd.cols * hd.rows)))
        {
                fprintf(stderr, "suppress : incorrect HIPS image input - size ?\n");
                exit(0);
        }

        float h_value, ang_tan;

        float *H       = new float [t_size];    // memory allocation
        float *T       = new float [t_size];
        float *data    = new float [t_size];
        float *val_trk = new float [t_size];
        float *rdg_trk = new float [t_size];

        pread(0, data, t_size*sizeof(float));   // read in HIPS data

        int x, y, curr_row = (hd.cols + 2);
        for (loop = 0; loop < h_size; loop++)   // check every pixel
        {
                y = loop % hd.cols;             // calc 2D coords from 1D data
                x = (loop - y) / hd.cols;

                if (curr_row != x) {            // per row progress report
                   curr_row = x;
                   if (loop % 10 == 0)
                      fprintf(stderr, "\n Suppressing Row : %3d", curr_row);
                   else
                      fprintf(stderr, " %3d", curr_row);
                }

                fscanf(h_fileptr, "%g\n", &H[loop]);    // get H value
                fscanf(t_fileptr, "%g\n", &T[loop]);    // get tan value

                h_value = H[loop];
                ang_tan = T[loop];

                //      first decide if pixel is a ridge or valley

                if (  ((h_value < threshold) && (h_value > (0 - threshold)))
                 ||   (h_value == -9)           // ignore "border" values
                 ||   (data[loop] == 0))        // ignore sea_level pixels
                {
                        rdg_trk[loop] = 0;      // non-candidate ridge point
                        val_trk[loop] = 0;      // non-candidate valley point
                        continue;               // next pixel in loop
                }

                // actually the angle of orientation is anticlockwise
                // from the y-axis, so by adding 90 degrees, we get
```



```
                // the angle clockwise from the x-axis, towards the
                // y-axis which determines which case holds.

                double angle = rad2deg(atan(ang_tan));
                angle += 90;

                if (angle < 0) angle += 180;            // same differences
                if (angle > 180) angle -= 180;

                double alpha, beta, tan_angle;

                /*
                 *      For non-maximal suppression,
                 *      there are four cases to be
                 *      considered :
                 *
                 *      [1] :   0 <= Angles <=  45
                 *      [2] :  45 < Angles <=  90
                 *      [3] :  90 < Angles <= 135
                 *      [4] : 135 < Angles <= 180
                 *
                 *      At each stage we calculate
                 *      two interpolated height values,
                 *      alpha and beta, which lie
                 *      along the line of curvature
                 *      represented by the tan of the
                 *      angle that it makes with the
                 *      x-axis.
                 *
                 *      For valley pixels (ie. H +ve)
                 *      the current pixel [x,y] must
                 *      be less than both alpha and
                 *      beta to survive suppression.
                 *
                 *      For ridge pixels (ie. H -ve)
                 *      the current pixel [x,y] must
                 *      be greater than both alpha and
                 *      beta to survive suppression.
                 *
                 *      The functions maximum and minimum
                 *      return either 0 and 1 corresponding
                 *      to a suppressed or non-suppressed
                 *      pixel.
                 */

                if (angle <= 45)                        // Case [1]
                {
                   tan_angle = tan(deg2rad(angle));

                   alpha = (1 - tan_angle) * data[loop - hd.cols] +
                        tan_angle * data[loop - hd.cols - 1];

                   beta = (1 - tan_angle) * data[loop + hd.cols] +
                        tan_angle * data[loop + hd.cols + 1];

                   if (h_value > 0)
                        val_trk[loop] = minimum(data[loop], alpha, beta);
                   else
                   if (h_value < 0)
                        rdg_trk[loop] = maximum(data[loop], alpha, beta);

                   continue;
                }

                if (angle <= 90)                        // Case 2 : 90 degs
                {
```

```
            angle = 90 - angle;
            tan_angle = tan(deg2rad(angle));

            alpha = (1 - tan_angle) * data[loop - 1] +
                    tan_angle * data[loop - hd.cols - 1];

            beta = (1 - tan_angle) * data[loop + 1] +
                    tan_angle * data[loop + hd.cols + 1];

            if (h_value > 0)
                val_trk[loop] = minimum(data[loop], alpha, beta);
            else
            if (h_value < 0)
                rdg_trk[loop] = maximum(data[loop], alpha, beta);

            continue;
        }

        if (angle <= 135)                       // Case 3 : 135 degs
        {
            angle = angle - 90;
            tan_angle = tan(deg2rad(angle));

            alpha = (1 - tan_angle) * data[loop + 1] +
                    tan_angle * data[loop - hd.cols + 1];

            beta = (1 - tan_angle) * data[loop - 1] +
                    tan_angle * data[loop + hd.cols - 1];

            if (h_value > 0)
                val_trk[loop] = minimum(data[loop], alpha, beta);
            else
            if (h_value < 0)
                rdg_trk[loop] = maximum(data[loop], alpha, beta);

            continue;
        }

        if (angle <= 180)                       // Case 4 : 180 degs
        {
            angle = 180 - angle;
            tan_angle = tan(deg2rad(angle));

            alpha = (1 - tan_angle) * data[loop - hd.cols] +
                    tan_angle * data[loop - hd.cols + 1];

            beta = (1 - tan_angle) * data[loop + hd.cols] +
                    tan_angle * data[loop + hd.cols - 1];

            if (h_value > 0)
                val_trk[loop] = minimum(data[loop], alpha, beta);
            else
            if (h_value < 0)
                rdg_trk[loop] = maximum(data[loop], alpha, beta);

            continue;
        }

        if (angle > 180)                // A problem - shouldn't arise
        {
                fprintf(stderr, "\n\nAngle > 180\n\n");
                continue;
        }
    }

    fclose(h_fileptr);                  // close H and tan files as
```



```
        fclose(t_fileptr);                          // were finished with them
        fprintf(stderr, "\n\n");

        /*
         *      Generate tracked HIPS image filenames
         *      and files. The filenames contain both
         *      a specifier indicating whether the
         *      HIPS image is of "tracked" valleys or
         *      ridges and the threshold value used
         *      (0.0) if none was indicated on the
         *      command line, along with the usual
         *      filename identifier used throughout, ie. :
         *
         *      identifier_val_0.05
         *      identifier_rdg_0.05
         */
        char valleys[40] = "";
        char ridges[40] = "";

        strcpy(valleys, file_id);
        strcat(valleys, "_val_");
        strcat(valleys, thresh_id);

        strcpy(ridges, file_id);
        strcat(ridges, "_rdg_");
        strcat(ridges, thresh_id);

        array2hips(val_trk, hd.cols, valleys, 1, 0);
        array2hips(rdg_trk, hd.cols, ridges, 1, 0);

        // free allocated memory

        delete H;
        delete T;
        delete data;
        delete rdg_trk;
        delete val_trk;
}
```

```
/*
 *      This code contains all the stack details
 *      used to track all minima or maxima in an
 *      image, removing isolated tracks with fewer
 *      than a minimum number of pixels.
 *
 *      usage : remove [-l int] < HIPS_track_image
 *
 *      The -l option allows the user to indicate
 *      a minimum length requirement. Tracks with
 *      less than this amount of pixels in them
 *      are deleted from the track image.
 */

#include <stdio.h>
#include <stdlib.h>
#include <hipl_format.h>

extern array2hips(float *, int, char[20], float, float);

char Progname[]="remove";               // needed by <hipl_format.h>

struct stacknode                        // definition of a stack node
{
        int coord;
        struct stacknode *nextptr;
};

typedef struct stacknode SNODE;
typedef SNODE *SNODEPTR;


/*
 *      push coordinate onto stack.
 */
void push(SNODEPTR *topptr, int coordinate)
{
        SNODEPTR newptr;

        newptr = malloc(sizeof(SNODE));

        if (newptr)
        {
                newptr->coord = coordinate;
                newptr->nextptr = *topptr;
                *topptr = newptr;
        }
        else
                fprintf(stderr, "No memory available for insertion.\n");
}


/*
 *      pop coordinate (returned) from stack.
 */
int pop(SNODEPTR *topptr)
{
        SNODEPTR tempptr;
        int popcoord;

        tempptr = *topptr;
        popcoord = (*topptr)->coord;
        *topptr = (*topptr)->nextptr;
        free(tempptr);
        return popcoord;
```



```
}


/*
 *      check emptiness of stack.
 */
int is_empty(SNODEPTR topptr)
{
        return !topptr;
}


/*
 *      print stack contents
 */
void print_stack(SNODEPTR currptr)
{
        if (currptr == NULL)
                fprintf(stderr, "Stack Empty.\n");
        else
        {
                fprintf(stdout, ":\n");

                while (currptr != NULL)
                {
                        fprintf(stdout, "%d ---> ", currptr->coord);
                        currptr = currptr->nextptr;
                }

                fprintf(stdout, "NULL\n\n");
        }
}


/*
 *      Free all memory held by a stack.
 */
void free_stack(SNODEPTR bogus_stack)
{
        int temp;

        while (!is_empty(bogus_stack))
        {
                temp = pop(&bogus_stack);
        }
}


/*
 *      This function pushes the 8 neighbouring
 *      coordinates of a point onto the stack.
 *       _________________________________________
 *      |             |         |                 |
 *      |  c - r - 1  |  c - r  |  c - r + 1      |
 *      |-------------|---------|-----------------|
 *      |    c - 1    |    c    |    c + 1        |
 *      |-------------|---------|-----------------|
 *      |  c + r + 1  |  c + r  |  c + r + 1      |
 *      |_____________|_________|_________________|
 */
void push_8_neighbours(SNODEPTR *currptr, int coord, int row)
{
        push(currptr, coord - row - 1);
        push(currptr, coord - row);
        push(currptr, coord - row + 1);
```

```
        push(currptr, coord - 1);
        push(currptr, coord + 1);

        push(currptr, coord + row - 1);
        push(currptr, coord + row);
        push(currptr, coord + row + 1);
}


/*
 *      This is the function which takes the array of
 *      pointers to the track stacks and produces the
 *      required HIPS image.
 */
void generate_track_image(SNODEPTR *track, int size)
{
        int number = 0;

        float * data = new float [size*size];

        while (!is_empty(track[number]))
        {
                while (!is_empty(track[number]))
                        data[pop(&track[number])] = 1;
                number++;
        }

        array2hips(data, size, "Removed", 1, 0);
}


/*
 *      This function builds up an array of all the
 *      tracks in an image which are at least min_length
 *      long. Each array element is in fact a
 *      pointer to a stack which contains all
 *      the coordinates of the points along
 *      that track :
 *
 *      array :   | track 1 | track 2 |          | track n |
 *                ----|---------|-------------------|------
 *                    |         |                   |
 *                 coord 1                         NULL
 *                    |
 *                    |
 *                 coord 2
 *                    |
 *
 *                    |
 *                 coord n
 *                    |
 *                    |
 *                   NULL
 */
void remove_tracks(char *array, int size, int min_length)
{
        SNODEPTR * track = new SNODEPTR [500];          // array of track
        SNODEPTR stackptr = NULL;

        //      push(&stackptr, value);
        //      pop(&stackptr);
        //      is_empty(stackptr);
        //      print_stack(stackptr);

        int coord, number = 0, tracklength;
```



```
        for (int c = 1; c < size - 1; c++)
          for (int r = 1; r < size - 1; r++)
          {
                tracklength = 0;                // length of current track
                track[number] = stackptr;       // specific track array element

                coord = r * size + c;
                if (array[coord] != 0) continue; // 0 => identified trackpt.

                array[coord] = -2;              // point now considered
                tracklength++;                  // increase track length
                push(&track[number], coord); // record coord on track stack
                                                // get the 8 neighbours
                push_8_neighbours(&stackptr, coord, size);

                while (!is_empty(stackptr))
                {
                        coord = pop(&stackptr);         // check next neighbour
                        if (array[coord] != 0) continue; // 0 => a trackpt.
                        array[coord] = -2;              // remove
                        tracklength++;                  // increase length
                        push(&track[number], coord); // record coord
                                                        // get 8 neighbours
                        push_8_neighbours(&stackptr, coord, size);

                }

                if (tracklength < min_length)
                        free_stack(track[number]);      // free memory
                else
                {
                        fprintf(stdout, "%d points on track ", tracklength);
                        print_stack(track[number]);

                        number++;               // begin next track in sequence
                }
          }

        fprintf(stdout, "Overall no. of track = %d\n", number);

        generate_track_image(track, size);
}


/*
 *      Main program begin.
 *
 *      Accepts minimum length as command line
 *      argument, otherwise it defaults to 0.
 */
main(int argc, char *argv[])
{
        int epsilon = 0;                        // default minimum length

        for (int loop = 1; loop < argc; loop++)
        {
                if (argv[loop][0] == '-')
                {
                    switch (argv[loop][1])
                    {
                        case 'l' :              // user supplied minimum length
                                sscanf(argv[++loop], "%d", &epsilon);
                                break;
                        default :               // command line arg error
                                fprintf(stderr, "usage : remove [-l int] < HIPS_track
_file\n");
```

```
                                exit(0);
                                break;
                        }
                }
        }

        struct header hd;
        read_header(&hd);                       // read image header details
        if (hd.pixel_format != PFBYTE)
        {
                fprintf(stderr, "remove: incorrect HIPS image input\n");
                exit(0);
        }
        int picsize = hd.cols * hd.rows;

        char * track_data = new char [picsize]; // allocate memory for data
        pread(0, track_data, picsize*sizeof(char));     // read in HIPS data

        fprintf(stdout, "Minimum Allowable Length = %d\n\n", epsilon);
        remove_tracks(track_data, hd.cols, epsilon); // remove small tracks

        delete track_data;                      // finished with it
}
```

C.4 Sundry

C.4.1 hipl_format.h

C.4.2 array2hips.cxx

```
/*
 * HIPL Picture Header Format Standard
 *
 * Michael Landy - 2/1/82
 */

struct header {
        char    *orig_name;     /* The originator of this sequence */
        char    *seq_name;      /* The name of this sequence */
        int     num_frame;      /* The number of frames in this sequence */
        char    *orig_date;     /* The date the sequence was originated */
        int     rows;           /* The number of rows in each image */
        int     cols;           /* The number of columns in each image */
        int     bits_per_pixel; /* The number of significant bits per pixel */
        int     bit_packing;    /* Nonzero if bits were packed contiguously */
        int     pixel_format;   /* The format of each pixel, see below */
        char    *seq_history;   /* The sequence's history of transformations */
        char    *seq_desc;      /* Descriptive information */
};

/*
 * Pixel Format Codes
 */

#define PFBYTE  0               /* Bytes interpreted as integers */
#define PFSHORT 1               /* Short int's interpreted as integers */
#define PFINT   2               /* Int's */
#define PFFLOAT 3               /* Float's */
#define PFCOMPLEX 4             /* 2 Float's interpreted as (real,imaginary) */
#define PFASCII 5               /* Ascii representation, with linefeeds after each ro
w */
#define PFQUAD1 11              /* quad-tree encoding */
#define PFBHIST 12              /* histogram of byte image */
#define PFSPAN  13              /* spanning tree format */
#define PLOT3D  24              /* plot-3d format */
#define PFAHC   400             /* adaptive hierarchical encoding */
#define PFOCT   401             /* oct-tree encoding */
#define PFBT    402             /* binary tree encoding */
#define PFAHC3  403             /* 3-d adaptive hierarchical encoding */
#define PFBQ    404             /* binquad encoding */
#define PFRLED  500             /* run-length encoding */
#define PFRLEB  501             /* run-length encoding, line begins black */
#define PFRLEW  502             /* run-length encoding, line begins white */
/* the following were added for the AI dept */
#define PFTRE   503             /* tracked edge format */
#define PFCCODE 504             /* line coding */

/*
 * Bit packing formats
 */

#define MSBFIRST 1              /* bit packing - most significant bit first */
#define LSBFIRST 2              /* bit packing - least significant bit first */

#define FBUFLIMIT 30000

/*
 * For general readability
 */

#define TRUE    1
#define FALSE   0

typedef int     Boolean;

/* extra bits tacked on in the AI dept */
```



```
/* text placement data */
struct text_print {int isize,ix,iy;} tp;

#ifdef IBMPC
#include <malloc.h>
#endif
```

```
/*
 *      Routine to take a data array of float values
 *      and scale the range from minimum : maximum
 *      into the required HIPS byte range 0 : 255
 *      and creates the correspondingly scaled HIPS
 *      image in the specified file.
 */

#include <stdio.h>
#include <hipl_format.h>

// char Progname[] = "array2hips";

/*
 *      Function which scales the values ranging
 *      from minimum : maximum into the 0 : 255 range.
 */
int scale(float value, float minimum, float maximum)
{
        return (int) ((value - minimum) / ((maximum - minimum) / 255));
}


/*
 *      Function which scales data into a bytesize
 *      HIPS image (values ranging from 0 : 255)
 *      factors depending on the maximum and minimum
 *      data value.
 */
void array2hips(float *data, int size, char file[20], float min, float max)
{
        unsigned char *bytearray;
        int i, sfd, picsize;
        struct header hd;

        picsize = size * size;

        if ((sfd = creat(file, 0666)) < 0)
        {
                fprintf(stderr, "array2hips : can't open output file %s\n", file);
                exit(0);
        }

        init_header(&hd,file,"",1,"",size,size,8,0,PFBYTE,"");
        fwrite_header(sfd, &hd);

        if ((bytearray = (unsigned char *) calloc (picsize, 1))
                        == (unsigned char *) NULL)
        {
                fprintf(stderr, "array2hips : can't allocate HIPS data space\n");
                exit(0);
        }

        for (i = 0; i < picsize; i++)                   /* scale orig data */
                bytearray[i] = scale(data[i], min, max);

        if (write(sfd, bytearray, picsize) != picsize)
        {
                fprintf(stderr, "array2hips : HIPS data write error\n");
                exit(0);
        }
}
```