Morphological Image Processing Operations: An Implementation

W. Martin

May 25, 2003

Updated: July 19, 2003

For student's own knowledge

Introduction

This paper attempts to go further than other papers on the web by including actual source code. As these filters are very simple to understand and simple to implement, I don't imagine there is a great need for a document such as this. However, if the need exists to check you work, then you may find this document useful. A warning is that the document in its current form implements the edge filtering using FFT convolution.

Most morphological operations are easy to perform on binary images. There has been much research on performing these algorithms on grayscale images. Typically, erosion and dilation perform fairly well under grayscale conditions. Functions that are based on hit-or-miss transformations, such as thinning and skeletonization, do not translate well for grayscale images.

1.0 Smoothing

Smoothing doesn't typically fall under the category of morphological operations. However, it is important as it can be used to promote outstanding results. Smoothing operators are used to reduce the noise in an image. By reducing the noise associated with an image, our results are much more "cleaner" and representative of the structure we are searching for.

1.1 Histograms

Histograms have many interesting applications in the field of image processing.

2.0 Edge Detection and Thresholding

In order to perform edge detection on an image, you must construct a kernel that can detect image discontinuities. Discontinuities are defined as a point, line, or edge in an image. By running a filter through the image, we define the points where the change in intensity matches some predefined threshold.

Edge detection algorithms are not just limited to a single gradient-based kernel. There are other relevant operations that are well-suited to particular operations.

Sobel Operator

Kirsch Operator

Prewitt Operator

Roberts Operator

Canny Edge Detection

Laplacian-of-Gaussian (LoG)

$$g(r) = \frac{1}{\sigma^2 2\pi} e^{-r^2/2\sigma^2}$$

In the simplest case (templated edge detection), the following matrix is used for the Laplacian:

-1	0	-1
0	4	0
-1	0	-1

Zero Crossing Filter

In most edge detection algorithms, there is a disadvantage in that the edges are not completely connected. It would be useful to have continuous edges for segmentation and further analysis. In order to accomplish this, we can use the Zero-Crossing filter. This filter uses the Laplacian as a starting point and puts an edge at every point where the Laplacian changes its sign (or crosses zero, which is the same). ([3] p.49)

Marr-Hildreth Edge Detection

Marr-Hildreth edge detection is a multi-scale edge detection method.

Watershed Detection

You'll notice in the images appendix that D isn't optimal for edge completeness, because there are no boundaries, thus we are ignorant as to where Lena's hair starts, for example. However it does perform the basic, brute force edge detection, thresholding, and dilation methods. It demonstrates the implementation of these functions, albeit not a demonstration of industry-ready algorithms. It must be noted that the thresholding that is done can be thought of as "blind thresholding", that is, it does not take into account neighboring or global pixel intensities and thus uses a constant to determine the cutoff intensity. From this argument, it is also noted that my method isn't the only *simple* method available for thresholding. One could use contrast stretching and inadvertently cause thresholding to occur. This would occur when selecting two points, (r1, s1) and (r2, s2), that control the contrast stretch. If one were to set r1=s1 and r2=s2, no change would occur in the image. However, if one were to set r1=r2, s1=0 and s2=L-1, a binary image would result.

Other types of thresholding techniques are:

Variable Thresholding (aka Adaptive Thresholding)

Band-thresholding

Multithresholding

Semithresholding

Hierarchical Thresholding

Multi-spectral

Thresholding also can occur in multi-spectral images.

3.0 Dilation and Erosion

Dilation and Erosion are processes in which we can grow or shrink discontinuities in an image by applying a mask.

For dilation, the mask used is

0	1	0
1	1	1
0	1	0

In morphological operations, this mask has a special name called a *structuring element*. In this case, the structuring element has the effect of dilating existing pixels, but giving them a "star-like" appearance. This is typically an unwanted outcome.

Usually, dilation is defined as a set operation.

Ultimate Erosion

4.0 Opening and Closing

5.0 Skeletonization

The process of skeletonization is reducing the objects in an image to a point where they are as thin as possible, but also retain their original shape. Thereby, they have the following properties: as thin as possible, connected, and centered.

There are many algorithms that have been devised for this. Here is a list of well-known and not-so-well-known algorithms:

(1) Hilditch's Algorithm
(2) Grassfire model
(3)

6.0 Watershed Algorithm

Constrained or conditional Watershed method.

7.0

8.0 Order-Statistics Filters

Although filters are not generally defined as morphological operations, I included them anyways. A order-statistics filter is a spatial filter that uses a ranking of the pixels in the neighborhood of the current pixel we are considering.

8.1 Median Filter

The median filter is an excellent filter for smoothing and reducing or eliminating certain types of noise. It works by calculating the median on its neighboring pixels and then setting itself to that value. Depending on the values of the "outliers", the median

filter performs better than the mean filter in that it preserves the detail better. Typically, salt and pepper noise can be completely removed from an image by using the median filter.

It works like such: we define the size of a neighborhood that we want to calculate the median over. This is typically 3x3. Larger dimensions can be used, at the effect of causing "blotchiness" in the image. However, the median filter can be used more than once to remove outliers that are hard to remove by just one pass. We apply the filter on each pixel in the image, by storing the intensity values, sorting them, and selecting the median, which in the 3x3 case is the value at index 5 (or 4, if using zero-based arrays) in the sorted vector list. This intensity value is applied to the current pixel. For the values at the border, we can choose to begin at the 2nd row and column and end at the width-1 and height-1. This may risk the fact that noise at the edges won't be handled so …

In Appendix A, there are three images; the original, salt and pepper noise added and the median filter applied to the image. It may be hard to note, but the third image is a little 'blurry' in comparison with the original. An adaptive filter could be devised that may help.

I used Matlab for this operation, with the source included in section 5.0 of Appendix B.

9.0 Application of Knowledge: Grain Boundary Detection

An initial image can present various origins and qualities. Therefore, some preliminary treatment may be necessary. There are many possible cases and solutions. Some of them are listed below:

- First, check if the image background is uniform. If not, you can apply any shade correction procedure.
- If the image is very noisy, it will be difficult to extract the grain boundaries and some filtering can help in further analysis. This seems to be a good practice to avoid, in this case, sharpening filters. You should try to use filters preserving edges, for example, a median filter.
- Thin, dark boundary lines can be effectively thickened by erosion but the whole image cannot be very noisy. Sometimes, better results are achieved using a top-hat transformation.
- In good quality images, with a limited amount of noise, the boundary lines can be effectively enhanced using edge detection filters. Unfortunately, this intuitively good solution only occasionally gives correct results.
- For some cases the best results can be obtained with the help of user-defined filters. This is, however, an advanced concept, which requires some experience.

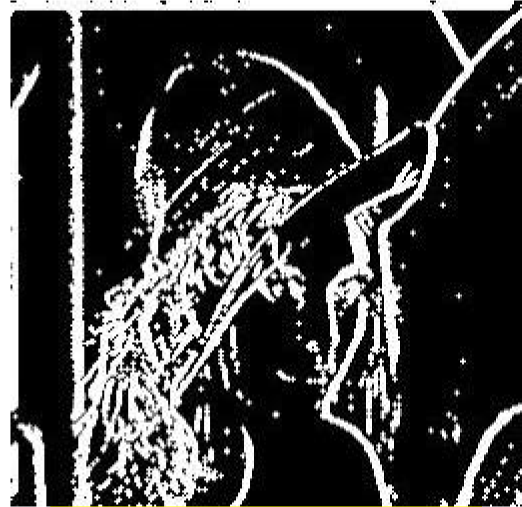
10.0 Appendix A: Images

Lena

Lena, grayscale – original image

Lena, edge-detected

Lena: Edge Detected and Thresholded



Lena: Edge, Thresholded, Dilation

Lizard with median filter applied.

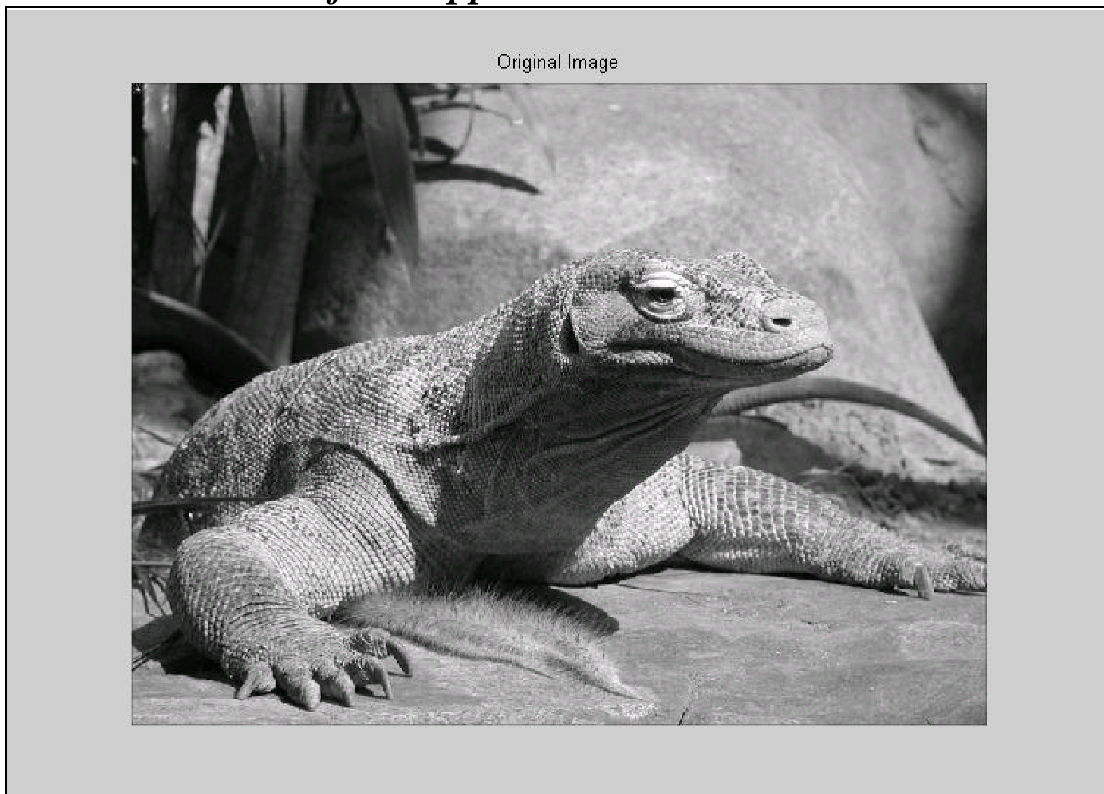


Figure 1.0: Original Image.

Figure 2.0: Lizard with Salt and Pepper noise added

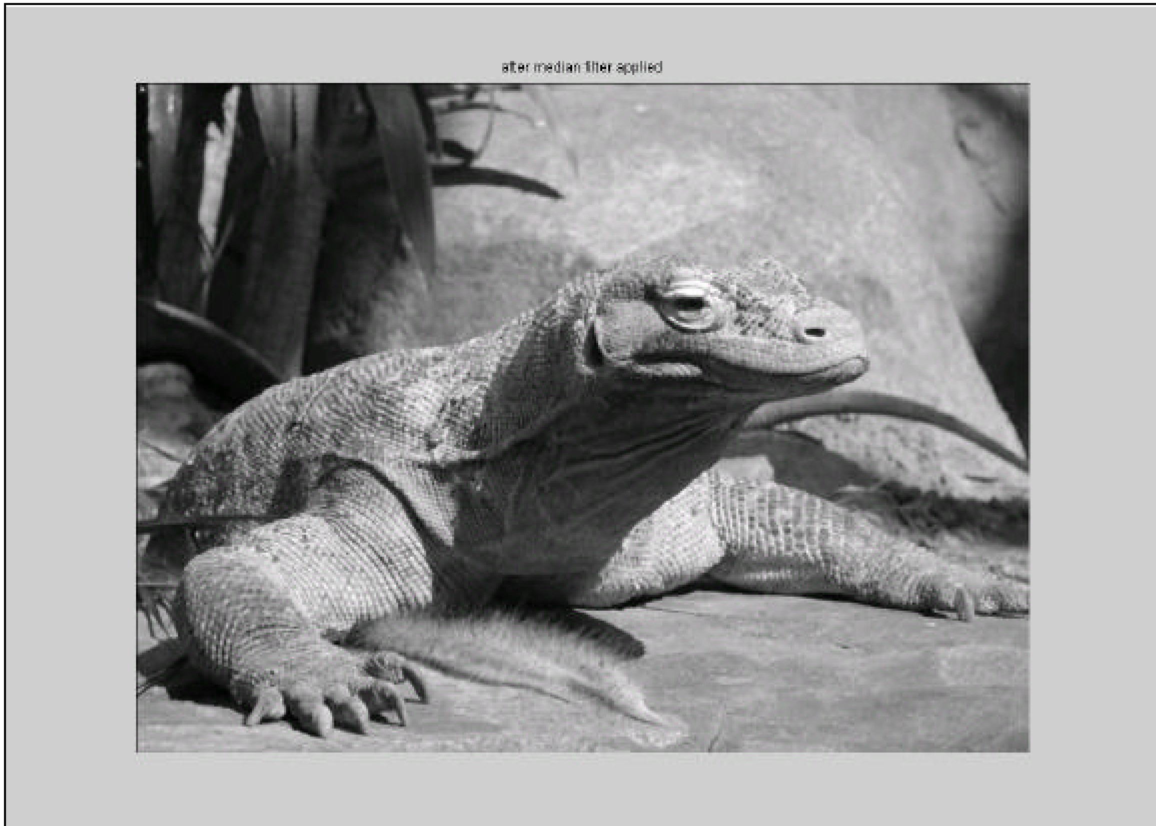


Figure 3.0 Lizard after median filter applied to figure 2.0. It's a little blurry after the filter is applied. An effect of the simple median filter.

Appendix B

1.0 Source Code for Edge Detection and Thresholding: thresh.c

```
/*******************************************************************
**       File name  : thresh.c
**       Written by : William Martin, Zhengpeng Li, Erin Nichols
**       Purpose    : to generate an image that highlights the edges
**                      of an image and perform thresholding on that
**                      image.
**      Usage      : thresh filename.pgm
**
**
*******************************************************************/
#include <stdio.h>
#include <math.h>
#include "pgm.h"
#include "fft2.h"


void ConvertToDouble(int h, int w, unsigned char* buftoconvert, double* buf);
void FreqMult(double* real, double* imag, double* real2, double* imag2,
                 int height, int width);

void DoEdge(double* buffer,int w);
void DoThresh(int height, int width, double* buffer, double thresh);

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

         int h,w;
         int i;
         int j;

         double* x_real;      // real component, image data
         double* x_imag;      // imaginary component, image data
         double* x_real2;     // real component, filter kernel.
         double* x_imag2;     // imaginary component, filter kernel
         char filename[80];  // filename creation buffer
         char tempstr[80];   // a temporary string

         unsigned char* bufptr; /* holds the image data */

         if(argc < 2 ) {

                  printf("Error: No parameter passed in. \n Usage: ./edge
filename.pgm \n");
                  return 0;
         }
         /////////////////////////////////////////////////
         // read in the image data.
         /////////////////////////////////////////////////
         bufptr =  read_image(argv[1],&h,&w);

         //////////////////////////////////////////////////////
         // x_real, x_imag: Freq. components of image to filter.
         //////////////////////////////////////////////////////
         x_real=(double *) malloc(h*w*sizeof(double));
         x_imag = (double *) malloc(h*w*sizeof(double));
```

```
        // Freq components for the edge filter
        ////////////////////////////////////////////////////
        x_real2 = (double*) malloc(h*w*sizeof(double));
        ////////////////////////////////////////////////////
        x_imag2 = (double*) malloc(h*w*sizeof(double));


        for(i = 0;i < h*w;i++) {
                x_imag[i] = 0.0;
                x_imag2[i] = 0.0;
        }

        //////////////////////////////////////////////////
        // must convert to double for the FFT algorithm.
        //////////////////////////////////////////////////
        ConvertToDouble(h, w, bufptr, x_real );

        //////////////////////////////////////////////////
        // build the filter kernel.
        //////////////////////////////////////////////////
        DoEdge(x_real2,w);

        //////////////////////////////////////////////////
        // 2-D FFT of image data and filter kernel
        //////////////////////////////////////////////////
        fft2(h, w, x_real,x_imag);
        fft2(h, w, x_real2,x_imag2);

        //////////////////////////////////////////////////
        // Multiply the frequency spectra
        //////////////////////////////////////////////////
        FreqMult(x_real, x_imag, x_real2, x_imag2, h , w);

        //////////////////////////////////////////////////
        // change it back to spatial domain for saving
        // to a file.
        //////////////////////////////////////////////////
        ifft2(h, w, x_real, x_imag);

        /////////////////////////////////////////////////
        // change image to "binary"
        /////////////////////////////////////////////////
        DoThresh(h, w, x_real, 15.0);

        for(j = 0;j < h*w;j+=w) {
                for(i = 0;i < w;i++) {
                        bufptr[j+i] = x_real[j+i];
                }
        }
        strcpy( tempstr, argv[1] );
        tempstr[strlen(tempstr)-4];

        for( i= 4; i < 80; i++ )
        if( tempstr[i] == '\0' )
                        tempstr[i-4] = '\0';

        sprintf( filename, "%s-thresh_seq.pgm", tempstr );
        write_image( filename, bufptr, w, h );

        return 0;
}
```

```
**      Function : ConvertToDouble
**      Purpose  : Converts an unsigned char array to
/*****************************************************************
**                double type.
**
*****************************************************************/
void ConvertToDouble(int h, int w, unsigned char* buftoconvert, double* buf) {

        int x;

        for(x=0;x<h*w;x++) {
                buf[x] = (double)buftoconvert[x];
        }
}

/*****************************************************************
**      Function: FreqMult
**      Purpose: To multiply two arrays in the frequency
**                domain.
**      Returns: real and imag hold the final result.
**                real2 and imag2 are only used for the
**                computation.
**      Equation:
**           temp = real(F) * real(H) - imag(F) * imag(H)
**           imag = imag(H) * real(F) + real(H) * imag(F)
**           real(F) = temp;
**
**        F = Frequency spectra of the image.
**        H = Frequency spectra of the filter kernel.
**        real = real component of freq. spectra
**        imag = imaginary component of the freq. spectra
**       temp = temporary hold, because we need last value
**                of real.
**
*****************************************************************/
void FreqMult(double* real, double* imag, double* real2, double* imag2,
                int height, int width) {
        int y = 0;
        double temp;

        for(y = 0;y < (width * height);y++) {
                temp = (double) (real[y] * real2[y] - imag[y] * imag2[y]);
               imag[y] = (double)(imag2[y]*real[y] + real2[y] * imag[y]);
                real[y] = temp;
        }
}

/*****************************************************************
**      Function:       DoEdge
**      Purpose :       to build the edge filter.
**
*****************************************************************/
void DoEdge(double* buffer,int w) {

        buffer[0]=-1.0/8.0;
        buffer[1]=-1.0/8.0;
        buffer[2]=-1.0/8.0;
        buffer[w]=-1.0/8.0;
        buffer[1+w]=-1.0/8.0;
        buffer[2+w]=1;
        buffer[2*w]=-1.0/8.0;
        buffer[1+2*w]=-1.0/8.0;
        buffer[2+2*w]=-1.0/8.0;
```

```
}

/**************************************************************
**      Function:       DoEdge
**      Purpose :       to build the edge filter.
**
**************************************************************/
void DoThresh(int height, int width, double* buffer, double thresh) {
        int y;
        for(y=0;y<(width*height);y++) {
                if(buffer[y]>thresh) {
                        buffer[y] = 255.0;
                }
                else {
                        buffer[y]=0.0;
                }
        }
}
```

fft2.h

```
#include <stdio.h>

void fft(int total_number, double *x_real, double *x_imag);
void ifft(int total_number, double *x_real, double *x_imag);
void fft2(int row_num, int col_num, double *x_real, double *x_imag);
void ifft2(int row_num, int col_num, double *x_real, double *x_imag);
```

fft2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

long N = 4;
const double PI = 3.1415926;

void fft(int total_number, double *x_real, double *x_imag);
void ifft(int total_number, double *x_real, double *x_imag);
void fft2(int row_num, int col_num, double *x_real, double *x_imag);
void ifft2(int row_num, int col_num, double *x_real, double *x_imag);
void reorder(int number, double *data);
void add_complex(double real1, double imag1, double real2, double
imag2, double *outreal, double *outimag);
```

```
void mul_complex(double real1, double imag1, double real2, double
imag2, double *outreal, double *outimag);
/////////////////////////////////////////////////////////////////////
////////
// fft2.c:      fft
// Purpose:     fft algorithm
// Parameters:  total_number - number of data to cal for all the
processors
//               x_real, x_imag - data for input and output
// Called by:   fft2
// Calls:        mul_complex
/////////////////////////////////////////////////////////////////////
////////

void fft(int total_number, double *x_real, double *x_imag)
{
  double tr, ti;
  double ur, ui;
  double sr, si;
  int stage;
  int i, j, k;
  int le, le2;
  int ip, jm1;
  int x;
  int ln;

  reorder(total_number, x_real);
  reorder(total_number, x_imag);

  ln = log(total_number)/log(2)+.5;
  le = 1;
  //loop for each stage
  for ( stage = 0; stage < ln; stage ++ )
  {
    le = pow(2, stage+1);
    le2 = le/2;

    ur = 1.0;
    ui = 0.0;
    sr = cos(PI/le2);
    si = (-1)*sin(PI/le2);

    for ( j = 1; j <= le2; j ++)
    {
      i = j - 1;

      while( i < N - 1 )
      {
        ip = i + le2;
        mul_complex(x_real[ip], x_imag[ip], ur, ui, &tr, &ti);
        x_real[ip] = x_real[i] - tr;
        x_imag[ip] = x_imag[i] - ti;
        x_real[i] = x_real[i] + tr;
        x_imag[i] = x_imag[i] + ti;
        i += le;
      }
```

```
        ur = tr*sr - ui*si;
        ui = tr*si + ui * sr;
        tr = ur;
      }//end j

    }

  }

  ///////////////////////////////////////////////////////////////////////
  ////////
  // fft2.c:      ifft
  // Purpose:     inverse fft algorithm
  // Parameters:  total_number - number of data to cal for all the
  processors
  //               x_real, x_imag - data for input and output
  // Called by:   fft2
  // Calls:        mul_complex
  ///////////////////////////////////////////////////////////////////////
  ////////

  void ifft(int total_number, double *x_real, double *x_imag)
  {
    double tr, ti;
    double ur, ui;
    double sr, si;
    int stage;
    int i, j, k;
    int le, le2;
    int ip, jm1;
    int x;
    int ln;

    reorder(total_number, x_real);
    reorder(total_number, x_imag);

    //use the conjugate as input
    for ( i = 0; i < total_number; i ++ )
    {
      x_imag[i] *= -1;
    }

    ln = log(total_number)/log(2) + 0.5;
    le = 1;
    //loop for each stage
    for ( stage = 0; stage < ln; stage ++ )
    {
      le = pow(2, stage+1);
      le2 = le/2;

      ur = 1.0;
      ui = 0.0;
      sr = cos(PI/le2);
      si = (-1)*sin(PI/le2);

      for ( j = 1; j <= le2; j ++)
```

```
      i = j - 1;

    { while( i < N - 1 )
      {
        ip = i + le2;
        mul_complex(x_real[ip], x_imag[ip], ur, ui, &tr, &ti);
        x_real[ip] = x_real[i] - tr;
        x_imag[ip] = x_imag[i] - ti;
        x_real[i] = x_real[i] + tr;
        x_imag[i] = x_imag[i] + ti;
        i += le;
      }
      tr = ur;
      ur = tr*sr - ui*si;
      ui = tr*si + ui * sr;

    }//end j

  }

  for ( i = 0; i < total_number; i ++ )
  {
    x_real[i] = x_real[i] / total_number;
    x_imag[i] = x_imag[i] / total_number;
  }



}

////////////////////////////////////////////////////////////////////////
////////
// fft.c:       fft2
// Purpose:     2-d fft algorithm using fft
// Parameters:  row_num - number of rows of the data array
//               col_num - number of cols of the data array
//               x_real, x_imag - data for input and output
// Called by:   main
// Calls:        fft
////////////////////////////////////////////////////////////////////////
////////

void fft2(int row_num, int col_num, double *x_real, double *x_imag)
{
  int i,j,k;
  double temp_real[row_num], temp_imag[row_num];

  N=col_num;
  //cal the col
  for ( i = 0; i < col_num; i ++ )
  {
    for ( j = 0; j < row_num; j ++ )
    {
      temp_real[j] = x_real[j*col_num + i];
      temp_imag[j] = x_imag[j*col_num + i];
    }
```

```
    fft(row_num, temp_real, temp_imag);

    for ( j = 0; j < row_num; j ++ )
    {
      x_real[j*col_num + i] = temp_real[j];
      x_imag[j*col_num + i] = temp_imag[j];
    }
  }

  //cal the row
  for ( i = 0; i < row_num; i ++)
  {
      for ( j = 0; j < col_num; j ++ )
      {
        temp_real[j] = x_real[i*row_num + j];
        temp_imag[j] = x_imag[i*row_num + j];
      }

    fft(col_num, temp_real, temp_imag);

    //only the master will collect the data
      for ( j = 0; j < col_num; j ++ )
      {
        x_real[i*row_num + j] = temp_real[j];
        x_imag[i*row_num + j] = temp_imag[j];
      }
  }

}

////////////////////////////////////////////////////////////////////////
////////
// fft.c:      ifft2
// Purpose:    2-d inverse fft algorithm using ifft
// Parameters: row_num - number of rows of the data array
//              col_num - number of cols of the data array
//              x_real, x_imag - data for input and output
// Called by:  main
// Calls:       ifft
////////////////////////////////////////////////////////////////////////
////////

void ifft2(int row_num, int col_num, double *x_real, double *x_imag)
{
  int i,j,k;
  double temp_real[row_num], temp_imag[row_num];

  N=col_num;
  //cal the col
  for ( i = 0; i < col_num; i ++ )
  {
    for ( j = 0; j < row_num; j ++ )
    {
      temp_real[j] = x_real[j*col_num + i];
      temp_imag[j] = (-1) * x_imag[j*col_num + i];
    }
```

```
      fft(row_num, temp_real, temp_imag);

      for ( j = 0; j < row_num; j ++ )
      {
        x_real[j*col_num + i] = temp_real[j]/row_num;
        x_imag[j*col_num + i] = temp_imag[j]/row_num;
      }
    }

    //cal the row
    for ( i = 0; i < row_num; i ++)
    {
        for ( j = 0; j < col_num; j ++ )
        {
          temp_real[j] = x_real[i*row_num + j];
          temp_imag[j] = x_imag[i*row_num + j];
        }

      fft(col_num, temp_real, temp_imag);

      //only the master will collect the data
        for ( j = 0; j < col_num; j ++ )
        {
          x_real[i*row_num + j] = temp_real[j]/col_num;
          x_imag[i*row_num + j] = temp_imag[j]/col_num;
        }
    }

  }

//////////////////////////////////////////////////////////////////////////
////////
// fft.c:        add_complex
// Purpose:      add two complex number together
// Parameters:  real1, imag1 - 1st complex number to add
//               real2, imag2 - 2nd complex number to add
//               *outreal, *outimag - output result
// Called by:   fft, ifft
// Calls:        none
//////////////////////////////////////////////////////////////////////////
////////

void add_complex(double real1, double imag1, double real2, double
imag2, double *outreal, double *outimag)
{
  *outreal = real1 + real2;
  *outimag = imag1 + imag2;
}

//////////////////////////////////////////////////////////////////////////
////////
// fft.c:        mul_complex
// Purpose:      multiply two complex number together
// Parameters:  real1, imag1 - 1st complex number to add
//               real2, imag2 - 2nd complex number to add
//               *outreal, *outimag - output result
```

```
// Calls:       none
////////////////////////////////////////////////////////////////////////
// Called by:  fft, ifft
////////

void mul_complex(double real1, double imag1, double real2, double
imag2, double *outreal, double *outimag)
{
  *outreal = real1 * real2 - imag1 * imag2;
  *outimag = real1 * imag2 + imag1 * real2;
}

////////////////////////////////////////////////////////////////////////
////////
// fft.c:      reorder
// Purpose:    bit reversal sorting the data
// Parameters: number - number of data in the array
//              x - array contains the data
// Called by:  main
// Calls:       none
////////////////////////////////////////////////////////////////////////
////////

void reorder(int number, double * x)
{
  int i,j,k;
  double temp;

  j = number/2;

  for ( i = 1; i < number-2; i ++)
  {
    if (i < j )
    {
      temp = x[j];
      x[j] = x[i];
      x[i] = temp;
    }
    k = number /2;

    while ( k <= j )
    {
      j = j - k;
      k = k /2;
    }
    j = j + k;
  }
  return;
```

2.0 Dilation and Erosion

In main, add:

```
////////////////////////////////////////////////////////////////////////////////
// be sure to initialize all items in dilate_imag above to 0.0 in the for loop
////////////////////////////////////////////////////////////////////////////////
DoDilate(dilate_real, w);
fft2(h, w, x_real,x_imag);
fft2(h, w, dilate_real, dilate_imag);

FreqMult(x_real, x_imag, dilate_real, dilate_imag, h , w);
ifft2(h, w, x_real, x_imag);


}/*******************************************************************
**      Function:      DoDilate
**      Purpose :      dilation.
**
**              0 1 0
**              1 1 1
**              0 1 0
******************************************************************/
void DoDilate(double* buffer, int w) {

        buffer[0]=0.0;
        buffer[1]=1.0;
        buffer[2]=0.0;
        buffer[w]=1.0;
        buffer[1+w]=1.0;
        buffer[2+w]=1.0;
        buffer[2*w]=0.0;
        buffer[1+2*w]=1.0;
        buffer[2+2*w]=0.0;
}
```

5.0 Median Filter

```
function y=Mediantest()
% function written by W.M.

    newimage = imread('lizbwsm.jpg');
    noisy = imnoise(newimage,'salt & pepper',0.02);
    figure(1);
    imshow(newimage);
    title('Original Image');

    figure(2);
    imshow(noisy);
```

```
    title('Salt and Pepper Noisy image');
    vr = noisy;
    % median filter applied here
    for x = 1:size(vr,1)-1
        for y = 1:size(vr,2)-1
            if(x > 1) && (y > 1)
                newarray = vr(x-1:x+1,y-1:y+1);
                newarray = newarray(:);
                newarray = sort(newarray,1);
                newv(x,y) = newarray(5);
            end
      end
   end
   figure(3);
   imshow(newv);
   title('after median filter applied');
end
```

References

[1] Woods, Richard, Gonzalez, Rafael, "*Digital Image Processing*", Second Edition, Prentice Hall, 2002

[2] Azar, Danielle, "Hilditch's Algorithm for Skeletonization", http://cgm.cs.mcgill.ca/~godfried/teaching/projects97/azar/skeleton.html

[3] Woznar, Leszek, "Image Analysis: Applications in Materials Engineering", 1999, CRC Press

[4] Kovesi, Peter, "Edges are not just steps", Department of Computer Science, University of Western Australia, January 2002, http://www.cs.uwa.edu.au/pub/robvis/papers/pk/ACCV62.pdf

[5] Baja, Thiel, "Skeletonization algorithm running on path-based distance maps", Imge and Vision Computing, March 1995, http://www.lim.univ-mrs.fr/~thiel/print/gsdb_thiel_ivc96.pdf

[6] Sonka, Milan, "Digital Image Processing: Chapter 5, Part 1 Segmentation:Thresholding", University of Iowa, Class 55:148 Digital Image Processing. Lecture Notes, http://www.icaen.uiowa.edu/~dip/LECTURE/Segmentation1.html