Study and Implementation of a Paper Currency Recognizer Using Cross-correlation Techniques

William Martin, May 2, 2003

1.0 Abstract

The objective of this paper is to complete the assigned homework #5. The student is directed to implement a currency recognition script in Matlab that will work with $1.00, $5.00, $10.00, and $20.00 paper currency images. Any kernel can be used to make this work.

2.0 Implementation

2.1 Choice of Kernel

In order to implement this algorithm, I chose a kernel that would be a distinguishing characteristic of a paper currency. This could be anything from the numbers in the corners, the words of the amount of the bill under the presidents' pictures, or the presidents faces themselves. The first two choices would probably work, but the DSP Guide for Scientists and Engineers recommends using the president faces. My choice, therefore, was using the president faces.

2.2 Cross-Correlation

Cross-correlation is the process of multiplying a known waveform by an "unknown" waveform to produce a third waveform that expresses similarities. A known waveform is defined, in our context, as a picture of some pattern that belongs to a certain classification. The unknown waveform is typically from a domain that we would expect certain objects to be found. For example, in the case of the defense industry, you may have an algorithm that takes a snapshot at a given instant by a CCD imaging device. If we are searching for a tank in the picture, we know that our domain will be the (1) outdoors and (2) most likely at or near the same level as the tank in relation to the ground. In the case of this assignment, our domain is much simpler case and only a subset of American currency is represented, and therefore we would expect to be able to come up with a match.

The equation for cross-correlation is **c [n] = a [n] * b [-n],** where a[n] is the unknown signal, and b[-n] is the known pattern, inverted. Multiplying these signals together produce a third signal, c [n]. The above equation implies convolution in the spatial domain. However, this assignment attempts to perform this calculation in the frequency domain.

When a proper cross-correlation occurs, the resulting signal has an impulse in a single location, while the rest of the picture is somewhat "flat". By negating the signal, a person can check for an absolute minimum, i.e. 0. That is the process that this paper and implementation attempt to demonstrate.

2.3 The Algorithm

I chose an equal kernel size for each bill of a width of 123 and a height of 164. The choice to pre-process the money.mat file to create separate images and store them into another file was done so because in reality, we wouldn't use the same images we are trying to classify as sources for the kernel at the time of classification. To illustrate this point, the idea of a tank recognition algorithm is again discussed. How do you create a filter kernel from an image of unknown patterns? You cannot do this, because the data in the image has not yet been classified!

In order to load the kernels into the memory workspace of Matlab, the program uses 'load presidents'. It then cuts apart the loaded kernel image file into smaller pieces in order to build the separate kernels that will participate in the cross-correlation algorithm. The algorithm proceeds to perform edge detection on each separate kernel, to increase the sharpness of the edges. This step produces better results. It also does so on the image to be classified, for the same reason. These operations occur in the frequency domain, thus a multiplication between the kernel and the edge detection matrix happens.

After we have performed the FFT and edge detection algorithms on all of the components, it is necessary to multiply the known pattern with the unknown pattern. Extracting the real part of the IFFT of the above operation, gives us the final matrix. This matrix needs to be scaled to be able to detect the impulse in the image.

2.4 Workarounds

As it seemed that I could not reach the true zeroes that I wanted to have in my images, another method had to be used; a method of discovering how to distinguish a matched pattern to an unmatched pattern in the resulting correlated images. The solution was that the pixel values in an image that was a pattern match, was that there was only one zero and the rest of the values were above some threshold. In the matched images, the next lowest value after 0 was in the range of 0.6 to 0.7, on a scale of 0.0 to 1.0.

I implemented a routine to sort the pixel values in increasing order. The sorted values allow me to look at the difference between consecutive pairs of pixel values. I used a threshold value of 0.11. If there is a difference of > 0.11 between the lowest number (in this case 0) and the next number, the value of 1 is returned, and the match is made. Otherwise, the entire set was discounted and it was treated as being a misclassification. No exact reasoning was found for zeroes appearing both in the known and unknown image. A few possibilities exist that may or may not explain it. For example, the sizes of the image and filter are not powers of 2. However, changing it to create images and filters to be powers of two will probably only add to the computation time. Another thought I had while

writing this paper was that the size and choice of the kernel affects the outcome. If you consider the example that was done in class (identifying the location of a number in a black/white image), the images didn't have a great amount of intensity changes, nor did it contain many different shades of gray. I came to this conclusion of filter kernels by discussing my problem with Alexsander. He, too, gets zeros but only around the outside boundary of his convolved image. He works around this problem by minimizing the size of the convolved image; he subtracts 10 pixels from each border.

2.5 Additional Information

On May 10, 2003, I discovered from reading a part of the Gonzalez and Woods' Digital Image Processing book that this method fails near the borders of the image, thus validating that this method doesn't work, as we believed it should. The original demonstration included a simple black and white image. Our pattern and pattern to match suffer the same problems as that in the book description. It is different than the demonstration given to us because of the nature of the location and color schemes used (b/w versus 256 levels of gray).

2.6 Conclusion

This assignment was an enjoyable assignment. One of the interesting aspects was trying to implement this in C for another class. I wasn't able to get the correct results, however I built most of the routines from scratch, including the FFT, convolution, and others. I will continue research and provide those results, if the professor is interested.

On the Matlab side, I was able to successfully implement a cross-correlation algorithm for recognizing paper money. What I learned from this assignment is that this is a poor method for any specialized, non-static imaging applications. This should be evident by looking at the picture of the 20-dollar bill. Old Hickory, aka Andrew Jackson, has a gauge in his head from where the currency was bent. Currency is usually not "bent" uniformly, so this method of correlation will not work, especially for badly "bruised" currency. It would work nicely, however, in a mint where new bills are produced.

With the problems and work-arounds, I felt like I was unsuccessful and successful at the same time. It appears everyone I've discussed the assignment with has gotten zeros in both known and unknown images. I can't speak for the entire class, but I discovered my own method that works consistently, no matter what the kernel choice is. Other implementations, such as decreasing the area of the convoluted image, as mentioned in my section on "work arounds", may work for only one kernel type and size. However, my error threshold value is dependent by the size and type of kernel, which requires empirical testing on the kernel to discover the error value beforehand.

The solution is successful in finding the pattern and classifying it correctly. The task given to this student was accomplished.

Appendix A-References

Smith, Steven W., "*The Scientist and Engineer's Guide to Digital Signal Processing*", 1997, California Technical Publishing, http://www.dspguide.com/

Chamberlain, Neil F. Dr., "*CENG 420, Design of Digital Signal Processing Systems*", SDSM&T, Spring 2003

Gonzalez, Woods, *"Digital Image Processing"*, 2001, Prentice Hall

Appendix B-Filter Kernel Picture

Appendix C-Source Code

Count Reps Function

Purpose: Counts the number of "zeros" in an image.

```
function y=count_reps(x)
% count_reps: get unique elements in x and count their frequency
% usage: u=count_reps(x)
%
% arguments:
%    x - any vector or array (arrays will be treated as a vector)
%    u - unique elements of x (sorted)
%    c - count of the number of times u(i) appeared in x
%
% note: can be used with either numeric or string data
% Author: William Martin, SDSM&T

u=sort(x(:));
n=length(u);
errval=0.11;
retval =0;
for i=1:n
    if(u(i)==0)
        retval=retval+1;
    end
    if(i>1)
        if(u(i)-u(i-1)>errval)
           y = retval;
            return;
        else
            retval = retval+1;
        end
    end


end
    y=retval;
end
```

Recognize_money Function

Purpose: To classify an image according to the pattern passed in.
Usage:

```
load money.mat
D=recognize_money (D01);  % will recognize D01.
D=recognize_money(flipud(D01)); % will also recognize D01
Other values of D05, D10, and D20 will work as well.
```

```
function y=recognize_money(im)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This function will find which bill type the file has.
% Sample usage: v=recognize_money(D01);
% Inputs: filename of paper currency to recognize.
```

```
% Assumptions about data: No extra filtering is needed to
% be able to classify the data.
% Written by William Martin, MS-CSC
% Original Author: Dr. Neil Chamberlain, Professor SDSM&T

Output: The value of the bill identified-("1","5","10","20")
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

amount = 0;
edge=[-1/8,-1/8,-1/8;-1/8,1,-1/8;-1/8,-1/8,-1/8]; % edge filter
% edge2 used for images (not the mask).
edge2=[-1/8,-1/8,-1/8;-1/8,1,-1/8;-1/8,-1/8,-1/8];

xi = (im2double(im));
xiFlipped=double(flipud(im));

Xi = fft2(xi);
XiFlipped=fft2(xiFlipped);

[r,c] = size(im);
edge2=fft2(edge2,r,c);
Xi=Xi.*edge2;
XiFlipped=XiFlipped.*edge2;

%create kernels
load('presidents');          % v now holds the president's heads
[presx,presy] = size(v);       % get the size of the entire thing

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Move on to the one dollar computation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
dollar = v(1:presx,1:presy/4);      % break up the presidents faces
dollars = imcomplement(flipud(fliplr(double(dollar))));
Dollars = fft2(dollars,r,c);
edgefft=fft2(edge,r,c);
Dollars=Dollars.*edgefft;

yDollars=Xi.*Dollars; % do the dollars first.
yDollarssf=real(ifft2(yDollars));

ynDollarssf=yDollarssf-min(yDollarssf(:));
ynDollarssf=ynDollarssf/max(ynDollarssf(:));

% count how many "true zeros" exist
ab = count_reps(ynDollarssf);

if (ab==1)  % do we have a perfect match?
    amount=1;
else    % no?  Then do the flipped computation
    yDollars=XiFlipped.*Dollars;
    yDollarssf=real(ifft2(yDollars));

    ynDollarssf=yDollarssf-min(yDollarssf(:));
    ynDollarssf=ynDollarssf/max(ynDollarssf(:));
    % check again for a "true" zero
    ab=count_reps(ynDollarssf);
    if(ab==1)
           amount=1;
     end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
end
% Move on to the five dollar computation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
fiver = v(1:presx,presy/4:presy/4*2); %
fivers = imcomplement(flipud(fliplr(double(fiver))));
Fivers = fft2(fivers,r,c);
Fivers=Fivers.*edgefft;

yFivers=Xi.*Fivers; % do the dollars first.
yFiverssf=real(ifft2(yFivers));

ynFiverssf=yFiverssf-min(yFiverssf(:));
ynFiverssf=ynFiverssf/max(ynFiverssf(:));

% count how many "true zeros" exist
ab = count_reps(ynFiverssf);

if(ab==1) % found a five
    amount=5;
else % check the upside down paper currency case.
    yFivers=XiFlipped.*Fivers;
    yFiverssf=real(ifft2(yFivers));
    ynFiverssf=yFiverssf-min(yFiverssf(:));
    ynFiverssf=ynFiverssf/max(ynFiverssf(:));
    ab = count_reps(ynFiverssf);

    if(ab==1)
        amount=5;
    end

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ten dollar computation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
tener = v(1:presx,presy/4*2:presy/4*3); % get the kernel
teners = imcomplement(flipud(fliplr(double(tener)))); % 180 rotate, invert
Teners = fft2(teners,r,c); % resize and get fourier spectrum.
Teners = Teners.*edgefft;    % apply edge filter

yTener=Xi.*Teners; % FFT Convolution
yTenerssf=real(ifft2(yTener)); % get only real part
ynTenerssf=yTenerssf-min(yTenerssf(:));
ynTenerssf=ynTenerssf/max(ynTenerssf(:));

% count how many "true zeros" exist
ab = count_reps(ynTenerssf);

if(ab==1) % found a ten
    amount=10;
else
    yTener=XiFlipped.*Teners;
    yTenerssf=real(ifft2(yTener));
    ynTenerssf=yTenerssf-min(yTenerssf(:));
    ynTenerssf=ynTenerssf/max(ynTenerssf(:));
    ab = count_reps(ynTenerssf);
    if(ab==1)
        amount=10;
    end
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% twenty dollar computation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
twenty = v(1:presx,presy/4*3:presy/4*4);

twentys = imcomplement(flipud(fliplr(double(twenty))));
Twentys = fft2(twentys,r,c);
Twentys=Twentys.*edgefft;

yTwenty=Xi.*Twentys; % do the dollars first.
yTwentyssf=real(ifft2(yTwenty));
ynTwentyssf=yTwentyssf-min(yTwentyssf(:));
ynTwentyssf=ynTwentyssf/max(ynTwentyssf(:));
% count how many "true zeros" exist
ab = count_reps(ynTwentyssf);

if(ab==1) % found a ten
    amount=20;
else
    yTwenty=XiFlipped.*Twentys; % do the dollars first.
    yTwentyssf=real(ifft2(yTwenty));
    ynTwentyssf=yTwentyssf-min(yTwentyssf(:));
    ynTwentyssf=ynTwentyssf/max(ynTwentyssf(:));
    ab = count_reps(ynTwentyssf);
    if(ab==1)
        amount=20;
    end
end
y = amount;
% end of function.
```

”Non-clean” C Version of Correlation Algorithm

Note: Not working. Just included for completeness (“humor”- including a non-complete version of source code, for “completeness”). I will attempt to complete at a later date. The original assignment required only a Matlab version, and thus all work that is extraneous is done for enjoyment and knowledge gain.

```
/*****************************************************************
*
 *      Written by: William Martin
**      purpose: to detect characters in images
**
*****************************************************************/
#include <stdio.h>
#include <math.h>
#include "pgm.h"
```

```
                  .01

void ConvertToDouble(int h, int w, unsigned char* buf, double* newbuf);
void FreqMult(double* real, double* imag, double* real2, double* imag2, int
height, int width);
unsigned char FindMin(int h, int w, unsigned char* buf);
unsigned char FindMax(int h, int w, unsigned char* buf);
int FindOneZero(int h, int w, unsigned char* buf);
void ShellSort( int array_size,unsigned char* buf);
void DoEdge(double* buffer,int w);
void dumpvec(int h, int w, unsigned char* buf);


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

        int h,w;
        int i;
        int j;

        unsigned char* argimage;
        unsigned char* dollarimage;
        unsigned char* fiveimage;
        unsigned char* tenimage;
        unsigned char* twentyimage;

        double* buffer;

       double* dollar_imaginary;
        double* fivebuffer;
        double* five_imaginary;
        double* tenbuffer;
        double* ten_imaginary;
        double* edge_real;
        double* edge_imag;

        double* twentybuffer;


        double* x_real;
        double* x_imag;
        double* finalbuf;

        double max = 0.0;
        double min = 2.0;
        printf("Trying to read files \n");

                            e(argv[1],&h,&w);

        dollarimage = read_image("george.pgm",&h,&w);
        fiveimage = read_image("abe.pgm",&h,&w);
        tenimage = read_image("alex.pgm",&h,&w);
        twentyimage = read_image("andy.pgm",&h,&w);


        buffer = (double *) malloc(h*w*sizeof(double));
        x_real = (double *) malloc(h*w*sizeof(double));
        dollarbuffer = (double *) malloc(h*w*sizeof(double));
                                          izeof(double));
        tenbuffer = (double *) malloc(h*w*sizeof(double));
        twentybuffer = (double *) malloc(h*w*sizeof(double));
```

```
edge_imag= (double *) malloc(h*w*sizeof(double));


dollar_imaginary = (double *) malloc(h*w*sizeof(double));
five_imaginary = (double *) malloc(h*w*sizeof(double));
ten_imaginary = (double *) malloc(h*w*sizeof(double));
twenty_imaginary = (double *) malloc(h*w*sizeof(double));


for(i = 0;i < h * w;i++)
{
        x_imag[i] = 0.0;
        edge_imag[i]=0.0;
}
printf("Convert to doubles \n");

ConvertToDouble(h, w,argimage, x_real);

ConvertToDouble(h,w,twentyimage,twentybuffer);
printf("calling fft \n");

fft2(h,w,x_real, x_imag);
fft2(h,w,twentybuffer,twenty_imaginary);
DoEdge(edge_real,w);
fft2(h,w,edge_real,edge_imag);
FreqMult(twentybuffer,twenty_imaginary, edge_real, edge_imag, h, w);
printf("Now Multiplying the kernels \n");

//                          ////////////////////////////
// multiply the filter kernel x currency image
////////////////////////////////////////////////////////
FreqMult(x_real,x_imag, twentybuffer, twenty_imaginary, h, w);

                                    eal[0]);

ifft2(h, w, x_real, x_imag);

printf("done IFFT,%g \n", x_real[0]);


for(j = 0;j< h * w;j+=w) {
        for(i = 0;i < w;i++) {
                argimage[j+i] = x_real[j+i]/255;
                printf("x-_real %g \n",x_real[j+i]/255.0);
        }
}

printf("Calling ShellSort \n");

min = FindMin(h, w, argimage);
max = FindMax(h, w, argimage);

for(i=0;i<h*w;i++) {
        argimage[i]=argimage[i]-min;

printf("Finding the min %d\n",min);

for(i=0;i<h*w;i++) {
        argimage[i]=argimage[i]/max;
}
printf("%d \n",FindOneZero(h,w,argimage));
```

```
        return 0;
}

/*************************************************
**      Function: ConverttoDouble
**      Purpose: To multiply two vectors
*************************************************/
void ConvertToDouble(int h, int w, unsigned char* buf, double* newbuf) {

        int x;

        for(x=0;x<h*w;x++) {
                newbuf[x] = (double)buf[x];
        }
}
/*************************************************
**      Function: FreqMult
**      Purpose: To multiply two vectors
*************************************************/
void FreqMult(double* real, double* imag, double* real2, double* imag2, int
heig
ht, int width) {

        int y=0;
        double temp;
                   idth*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: FindMin
**
*************************************************/
unsigned char FindMin(int h, int w, unsigned char* buf) {

        int i;
        unsigned char min=255;
        for(i=0;i<h*w;i++)
                if(buf[i]<min) min=buf[i];
        }
        return min;

}
/                 ************************************
**      Function: FindMax
**
*************************************************/
unsigned char FindMax(int h, int w, unsigned char* buf) {

        int i;
        unsigned max = 0;
        for(i=0;i<h*w;i++) {
                if(buf[i]>max) max=buf[i];
        }
        return max;
```

```
                t h, int w, unsigned char* buf) {
        int i;
        int lowest0=0;
        for(i=0;i<h*w;i++) {

                        printf("buf[%d] = %d \n",i,buf[i]);

        }

}


/*****************************************************
**      Function : FindOneZero
**      Function : to find zeros
*****************************************************/
int FindOneZero(int h, int w, unsigned char* buf) {

        int zeros=0;
        int i=0;

        for(i=0;i<h*w;i++)  {
                if(buf[i] < ERROR_TERM) {
                        zeros++;
                }
        }
        return zeros;
}

/*****************************************************
**      Function : DoEdge
**      Functio                                    rix
*****************************************************/
void DoEdge(double* buffer,int w) {

        buffer[0]=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;



}
```

PGM Source Code: PGM.C

```
#include "ppm.h"

  ////////////
 // ppm.c //
/                          ////
 // copyright Erin Nichols, 2003 //
////////////////////////////////////////////////////////////////////////
 // routines for reading, writing, and deallocating memory for PGM images //
/  ////////////////////////////////////////////////////////////////////

  ////////////////
 // read_image //
//////////////////////////////////////////////////
 // Reads an entire PGM-format image all at once //
//////////////////////////////////////////////////
    // Requirements for PGM file:          //
   // the magic P5 is before any comments //
  // comments are < 80 characters        //
 // max val = 255                       //
//////////////////////////////////////////
/* Parameters:
 * filename - full name of the image file.
 * width - width of image in pixels
 * height - height of image in pixels
 */
unsigned char *read_image(char *filename, int *width, int *height)
{
    FILE *inptr;  // input file pointer
    int max;      // max val, should be 255.
    long bufsize; // size of buffer in bytes
    unsigned char *buf; // the image buffer
    int done;     // used as boolean for loop exit
    int temp;     // temp integer
    char tempstring[81]; // temp string for fscanf

    /*---------- open file ----------*/
    inptr = fopen(filename,"r");
    if(!inptr)
    {
        fprintf(stderr, "Error opening file.1 %s\n", filename );
        return 0;
    }

    /*---------- read header ----------*/
    //fscanf( inptr, "%s %d %d %d ", tempstring, width, height, &max );

    // P5
    fscanf( inptr, "%s", tempstring );

    if( strcmp( tempstring, "P5" ) )
    {
       fprintf( stderr, "Image must be in P5 PGM format with maximum value
255.\n");
       return NULL;
```

```
    // width

    while( !done )
    {
        fscanf( inptr, "%s", tempstring );

        if( tempstring[0] == '#' )
            fscanf( inptr, "%80[^\n]", tempstring );
        else
        {
            if( isdigit( tempstring[0] ) )
            {
                sscanf( tempstring, "%d", width );
                done = 1;
            }
            else
            {
                fprintf( stderr,
                    "Image must be in P5 PGM format with maximum value
255.\n");
                return NULL;
            }
        }
    }


    // height
    done = 0;
    while( !done )
    {
        fscanf( inptr, "%s", tempstring );

        if( tempstring[0] == '#' )
            fscanf( inptr, "%80[^\n]", tempstring );
        else
        {
            if( isdigit( tempstring[0] ) )
            {
                sscanf( tempstring, "%d", height );
                done = 1;
            }
            else
            {
                fprintf( stderr,
                    "Image must be in P5 PGM format with maximum value
255.\n");
                return NULL;
            }
        }
    }


    // max
    done = 0;
    while( !done )
    {
        fscanf( inptr, "%s", tempstring );

        if     pstring[0] == '#' )
            fscanf( inptr, "%80[^\n]", tempstring );
        else
```

```
            if( isdigit( tempstring[0] ) )
            {
                    nf( tempstring, "%d", &max );
                done = 1;
            }
            else
            {
                fprintf( stderr,
                    "Image must be in P5 PGM format with maximum value
255.\n");
                return NULL;
            }
        }
    }
    if( max != 255 )
    {
        fprintf( stderr, "Image must be in P5 PGM format with maximum value
255.\n");
        return NULL;
    }

    // read whitespace
    fscanf( inptr, " " );


    /*---------- Allocate buffer ----------*/
    bufsize = (*width) * (*height) * sizeof(unsigned char);
    buf = (unsigned char *) malloc( bufsize );
    if( !buf )
    {
        fprintf( stderr, "Error allocating memory for image.\n" );
        return NULL;
    }


    /*---------- Read image data ----------*/
    if( fread(buf, 1, bufsize, inptr) != bufsize )
    {
        fprintf(stderr, "Error reading from %s\n", filename );
        return NULL;
    }

    fclose(inptr);

    return buf;

} /* read_image */


  /////////////////
 // write_image //
/////////////////////////////////////////////////////
 // Writes an entire PGM-format image all at once //
/////////////////////////////////////////////////////
/* Paramet
 * filename - full name of the image file.
 * buf - the image buffer
 * width - width of image in pixels
 * height - height of image in pixels
 */
int write_image(char *filename, unsigned char *buf, int width, int height)
```

```
    FILE *outptr; // output file pointer
    long bufsize; // size of buffer in bytes

    /*---------- open file ----------*/
    outptr = fopen(filename,"w");
    if(!outptr)
    {
        fprintf(stderr, "Error opening file %s\n", filename );
        return 0;
    }

    /*---------- write header ----------*/
    fprintf( outptr, "P5\n%d %d\n255\n", width, height );

    /*---------- Write image data ----------*/
    bufsize = width * height * sizeof(unsigned char);

    if( fwrite(buf, 1, bufsize, outptr)  != bufsize )
    {
        fprintf( stderr, "Error writing to %s\n", filename );
        return 0;
    }

    fclose(outptr);

    return 1;
} /* write_image */


  /////////////////
 // free_image //
////////////////////////////////////////////////
 // Frees memory allocated to an image buffer //
////////////////////////////////////////////////
void free_image( unsigned char *buf )
{
    if( !buf )
        return;
    free( (void*)buf );

    buf = NULL;
}
```

PGM.H

```
/* PGM code -- Erin Nichols -- 2003
 *
 * Methods for reading, writing, allocating and deallocating memory for
 * PGM images.
 *
 * Requirements for PGM files:
 * PGM files must meet the PGM standard, and:
 * - Max Val must be 255. In practice, most applications won't create
 *   images with less than 255 since anything up to 255 requires one byte
```

```
 *   per channel per pixel anyway. Applications won't use more than 255
 *   because that was the limit on the old standard. The new standard's maximum
 *   is 65536.
 * - The magic characters "P5" must be at the front of the file, ie, before any
 *   comments. The PGM man page doesn't say this is part of the standard, but I
 *   suspect it is. Either way, applications don't put comments before the "P5"
 *   in practice.
 * - Comments must be < 80 characters. This is generally true in practice, but
 *   not part of the standard.
 *
 * Other Notes:
 *
 * Unsigned characters are one byte, so this is the data type used for the
 * image buffer.
 *
 *
 */

/*---------- Included files ----------*/
#include <stdio.h>

/*---------- Prototypes ----------*/
unsigned char * read_image( char* filename, int* width, int* height );
int write_image( char* filename, unsigned char* buffer, int width, int height);
void free_image( unsigned char* filename );
```

FFT.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, doubl

////////////////////////////////////////////////////////////////////////////
// fft2.c:      fft
// Purpose:     ff
// Parameters:  total_number - number of data to cal for all the processors
//              x_real, x_imag
//
// Calls:       mul_complex
///              ////////////////////////////////////////////////////////////

void fft(int tota
{
```

```
  double ur, ui;
  double sr, si;

  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;
      }
      tr = ur;
      ur = tr*sr - ui*si;
      ui = tr*si + ui * sr;



  }

}


////////////////////////////////////////////////////////////////////////////
// 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:     t2
// Calls:       mul_complex
////////////////////////////////////////////////////////////////////////////

void ifft       tal_number, double *x_real, double *x_imag)
{
```

```
  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
                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;
  N=row_num;
  double temp_real[row_num], temp_imag[row_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
```

```
////////////////////////////////////////////////////////////////////////////


{
  int i,j,k;
  N=row_num;
  double temp_real[N], temp_imag[N];


  //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;
```

```
}

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

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
////////////////////////////////////////////////////////////////////////////////

v
{
  int i,j,k;
  double temp;

  j =

  for ( i = 1; i < number-2; i ++)
  {
    if (i < j )

      temp = x[j];
      x[j] = x[i];
      x[i] = temp;
    }
    k =

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

FFT.H

```
////////////////////////////////////////////////////////////////////////////
// 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)
{
[root@localhost recognizer]# cat 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);
```