Image Library  Version 2

C/C++ Image Processing Routines for EE211A

Documentation prepared by Matthew Fong (mattfong@icsl.ucla.edu)

The ANSI/ISO C/C++ routines in the library image_lib.c are intended to illustrate concepts introduced in image processing by implementing commonly used image transforms such as the FFT and DCT. There are functions that gather statistics on images, including the energy, mean and standard deviation. Programming is made easier by providing commonly used functions such as I/O in a simple to use routines, thus hiding the complexity behind the system calls that need to be made. These routines form a basic foundation for which more complex image processing techniques may be examined.

The following classes of routines are provided:

- Input/Output
- Transforms
- Filtering
- Image operations
- Image statistics

The rest of this paper is dedicated to documenting the individual routines each classes. Of important note, these functions assume that square images with lengths of powers of two are used. The format of the images must also be in GRAY format. ImageMagick™, a conversion utility available for Windows 95/98/NT, UNIX, Linux and the MacOS, allows the GRAY format to be converted to other formats such as TIFF and BMP to be viewed under various platforms. Images may also be converted to GRAY using this utility as well. It may be found at http://www.wizards.dupont.com/cristy/ImageMagick.html. To avoid viewing or conversion problems, a GRAY file should have .gray as its file extension.
Input / Output

Main routines:

void read_image(unsigned char* header, unsigned char** image, char* filename, int n)
void write_image(unsigned char* header, unsigned char** image, char* filename, int n)
void write_float_values(float** image, char* filename, int n)
void generate_header(unsigned char* header, int x_dim, int y_dim) [bugs are present]

Associated routines:

void byte2float(unsigned char** array_in, float** array_out, int n)
void float2complex(float** array_in, Complex** array_out, int n)
void float2byte(float** array_in, unsigned char** array_out, float scale, int n)
void array_abs_val(Complex** array_in, float** array_out, float* max_val, float* avg_val, int n)

The images used for these routines are in GRAY format. This is a grayscale image format that stores each pixel using 8-bits, thus having a range from 0 to 255. Zero denotes a black pixel, 255 denotes a white pixel and intermediate values are shades of gray.

Main routines:

void read_image(unsigned char* header, unsigned char** image, char* filename, int n)

Input: unsigned char* header, unsigned char** image, char* filename, int n
Output: unsigned char* header, unsigned char** image

This function takes the filename of an image in GRAY format and reads it into a two-dimensional unsigned character array image. This preserves the 8-bit representation of each pixel. The length of the side of an image is provided by the argument n (the image is assumed to be square). A 256-bit header precedes the actual image data and must be stripped out before the image is read and is stored in the vector header. This header information is needed when the image is written back out to disk.

void write_image(unsigned char* header, unsigned char** image, char* filename, int n)

Input: unsigned char* header, unsigned char** image, char* filename, int n

This function takes a two-dimensional unsigned character array image and writes it to a file in GRAY format. The length of the side of an image is provided by the argument n (the image is assumed to be square). A 256-bit header stored in the vector header precedes the image data and must be written first.

void write_float_values(float** image, char* filename, int n)

Input: float** image, char* filename, int n
This function takes a two-dimensional float array `image` and writes the values of the array into a file, ordered by row. The filename must be supplied through the argument `filename`. The length of the side of an image is provided by the argument `n`.

```c
void generate_header(unsigned char* header, int x_dim, int y_dim)
Input: unsigned char* header, int x_dim, int y_dim
Output: unsigned char* header
```

This function generates a header for an image with dimensions `x_dim` by `y_dim` and is usually used when creating an image with a C-program. The header from the original file should always be used.

**Associated routines:**

```c
void byte2float(unsigned char** array_in, float** array_out, int n)
Input: unsigned char** array_in, float** array_out, int n
Output: float** array_out
```

```c
void float2byte(float** array_in, unsigned char** array_out, float scale, int n)
Input: float** array_in, unsigned char** array_out, float scale, int n
Output: unsigned char** array_out
```

```c
void float2complex(float** array_in, Complex** array_out, int n)
Input: float** array_in, Complex** array_out, int n
Output: Complex** array_out
```

```c
void array_abs_val(Complex** array_in, float** array_out, float* max_val, float* avg_val, int n)
Input: Complex** array_in, float** array_out, float* max_val, float* avg_val, int n
Output: float** array_out
```

The associated routines `byte2float()`, `float2byte()` and `float2complex()` convert the image from the current type to the specified type. All images that are read using `read_image()` are returned as type `unsigned char`. Most routines require that the input data be of type float or Complex, so the appropriate image routine must be used. The imaginary part of an image is initialized to zero when `float2complex()` is called. To convert an image of type Complex to float, the absolute value of the array must be taken using the routine `array_abs_value()`. The maximum value and average value of the image is returned through the referenced values `max_val` and `avg_val`. Take careful note that `float2byte()` also requires a scale value `scale` as input. After taking an image transform such as the FFT, it is necessary to scale the output array so that the correct image can be seen. The conversion is accomplished by multiplying values in `array_in` by `scale` and rounding to the nearest integer, resulting in a linear scale. Values falling outside the interval [0, 255] are clipped to 0 and 255. Since the maximum pixel value is 255, the scale value is usually 255/max_val. However, a better normalization involves traversing the entire image and applying a logarithmic scale, as follows:

\[
image[m][n] = \left(\frac{255.0}{c1}\right) \times (c1 + \log10(image[m][n])), c1 \text{ a constant taking on a value in (1, 10]}
\]

This logarithmic scale result has much better contrast. Note that the `scale` argument is 1.0 if the latter method is used. If no transform is applied, the `scale` value is simply one. The user must call the above routines.
**Transforms**

**Main routines:**
- `int FFT2D(Complex** image, int n, int direction)`
- `int FFT(int dir, int m, double* x, double* y)`
- `int DCT2D(float** array, int n, int direction)`
- `int DCT(int dir, int m, int n, float* array)`

**Associated routines:**
- `void array_abs_val(Complex** array_in, float** array_out, float* max_val, float* avg_val, int n)`
- `void shift_2d_float(float** array, int n)`
- `int Powerof2(int n, int* exponent)`
- `void Complex_multiplication(Complex a, Complex b, Complex* result)`

Image transforms such as the Fourier Transform or Discreet Cosine Transform are extremely useful. The Fourier Transform looks at the frequency content of an image and allows operations such as filtering. The Discreet Cosine Transform is used in many image compression algorithms, a well-known application being JPEG. The one-dimensional as well as two-dimensional routines of both are provided.

**Main routines:**

`int FFT2D(Complex** image, int n, int direction)`

**Input:** Complex** image, int n, int direction

**Output:** Complex** image, 1 (success) or 0 (failure)

This function performs an in-situ 2D FFT given a two-dimension Complex array `image` as input. The size of the image must be provided through the argument `n`. Only square images with sizes that are powers of two can be used, otherwise the function will return an error. The argument `direction` is used to denote a forward or inverse transform, with 1 denoting forward, and -1 denoting inverse. Upon completion, `FFT2D()` returns a one to indicate success. If an error is encountered when the image supplied is square and whose length is a power of two, a memory allocation error has occurred. Note that the indexing should be done according to the DFT convention, with the (0,0) element being the element at the upper left corner of the image.

The actual FFT is not taken by `FFT2D()`, but rather it calls `FFT()`. Since the kernel of the Fourier Transform is separable, this allows the image to transformed by its rows and then its columns. The resulting time savings is enormous for large images.

`int FFT(int dir, int m, double* x, double* y)`

**Input:** int dir, int m, double* x, double* y

**Output:** double* x, double* y, 1 (success)
This function computes an in-situ one-dimensional unitary complex-to-complex FFT. The one-dimensional double arrays \( x \) and \( y \) are the real and imaginary arrays of the corresponding row or column of a Complex image. They are both of length \( n = 2^m \) points. The value of \( m \) is determined using the function \texttt{Powerof2}(). The argument \texttt{dir} specifies a forward or inverse transform, where 1 gives the forward transform and -1 gives the inverse transform. Upon completion, \texttt{FFT()} returns a one to indicate success.

\textbf{int DCT2D(float** image, int n, int direction)}

\textbf{Input:} float** image, int n, int direction  
\textbf{Output:} float** image, 1 (success) or 0 (failure)

This function performs an in-situ 2D DCT given a two-dimension float array (not Complex) \texttt{image} as input. The size of the image must be provided through the argument \( n \). Only square images with sizes that are powers of two can be used, otherwise the function will return an error. The argument \texttt{direction} is used to denote a forward or inverse transform, with 1 denoting forward, and -1 denoting inverse. Upon completion, \texttt{DCT2D()} returns a one to indicate success. If an error is encountered when the image supplied is square and whose length is a power of two, a memory allocation error has occurred.

As with \texttt{FFT2D()}, the actual DCT is not taken by \texttt{DCT2D()}, but rather it calls \texttt{DCT()}. Since the DCT can be computed using the FFT, \texttt{FFT()} is called during the computation of the DCT. This results in a fast DCT algorithm.

\textbf{Note:} After taking the forward 2D DCT, the resulting matrix of values will contain positive and negative values. Normalizing the image to the range \([0, 255]\) will require adding the absolute value of the largest negative value to every element of the matrix using the function \texttt{add_scalar}(). The largest negative value can be found with the function \texttt{maximum_value()} with the \texttt{positive_or_negative} argument set to 0. Once this is done, the maximum value \texttt{max_val} must be found using the function \texttt{maximum_value()} with the \texttt{positive_or_negative} argument set to 1. The image can then be linearly normalized using \( 255/\texttt{max_val} \) in the function \texttt{float2byte}(). However, a better normalization involves traversing the entire image and applying a logarithmic scale, as follows:

\[
\text{image}[m][n] = (255.0 / c1) \times (c1 + \log10(\text{image}[m][n])), \ c1 \text{ a constant taking on a value in } (1, 10)
\]

\textit{If these method results in the truncation of too many elements to zero, then output the non-normalized image to a file using \texttt{write_float_values()} and view the results in MATLAB.}

\textbf{int DCT(int dir, int m, int n, float* array)}

\textbf{Input:} int dir, int m, int n, float* array  
\textbf{Output:} float* array, 1 (success) or 0 (failure)

This function computes an in-situ one-dimensional unitary DCT. The one-dimensional float argument \texttt{array} corresponds to the row or column of a float image. It is of length \( n = 2^m \) points. The value of \( m \) is determined using the function \texttt{Powerof2}(). The argument \texttt{dir} specifies a forward or inverse transform, where 1 gives the forward transform and -1 gives the inverse transform. Upon completion, \texttt{dct()} returns a one to indicate success, otherwise zero denotes that memory allocation errors occurred.
Associated routines:

void array_abs_val(Complex** array_in, float** array_out, float* max_val, float* avg_val, int n)

  Input: Complex** array_in, float** array_out, float* max_val, float* avg_val, int n
  Output: float** array_out

After performing a FFT on an image, it will be necessary to convert it from type Complex to float in order to output the image. The maximum value and average value of the image is returned by reference through the arguments max_val and avg_val, respectively. The user must call this function.

void shift_2d_float(float** array, int n)

  Input: float** array, int n
  Output: float** array

When the FFT of an image is taken, the DC component is at the corners of the resultant image when the inverse transform is displayed. To display the image correctly, the second and fourth quadrants must be swapped and the first and third quadrants must be swapped. This is due to the DFT indexing convention, which places (0, 0) element not at the center of the image, but at the upper left corner. This function should be applied after the FFT of an image is taken. The user must call this function to ensure the image is properly shifted.

int Powerof2(int n, int* exponent)

  Input: int n, int* exponent
  Output: int* exponent, 1 (success) or 0 (failure)

This function checks to make sure that the length of the sides of a square image are a power of two, that is \( n = 2^{\text{exponent}} \). It is called from within fft2d() and dct2d(). The argument n provides the length of a side of the image. If n is a power of two, the appropriate exponent is returned by reference and one is returned to indicate success. Otherwise, zero is returned to indicate that n is not a power of two.

void Complex_multiplication(Complex a, Complex b, Complex* result)

  Input: Complex a, Complex b, Complex* result
  Output: Complex* result

Complex multiplication is needed in the computation of the DFT and is called by dct(). This function takes two Complex values a and b and multiplies them together. The result of the multiplication is passed by reference through the argument result.
Filtering

Main routines:

void Zonal_LP(Complex** image_complex, float rho_0, int n)
void Zonal_HP(Complex** image_complex, float rho_0, int n)
void Butterworth_LP(Complex** image_complex, float rho_0, float order, int n)
void Butterworth_HP(Complex** image_complex, float rho_0, float order, int n)
void median_filter(float** image_corr, float** med_img, int median_3_or_5, int n)

Associated routines:

int FFT2D(Complex** image, int n, int direction)
void shift_2d_complex(Complex** array, int n)
float min3(float a, float b, float c)
float median3(float a, float b, float c)
float median5(float a, float b, float c, float d, float e)

A consequence of the Fourier Transform is transform domain filtering. The basic idea is to take the transform of an image, multiply the image and filter together, and take the inverse transform. Zonal and Butterworth filters are implemented in this library. Zonal filters have sharp cutoffs, unlike Butterworth filters, which have a smooth roll-offs. The sidelobes of the Butterworth filter lead to an observable phenomenon known as **ringing**. Median filtering is not transform domain filtering but is used to improve images with corruptive noise.

Main routines:

void Zonal_LP(Complex** image_complex, float rho_0, int n)

Input: Complex** image_complex, float rho_0, int n

Output: Complex** image_complex

This routine takes an in-situ zonal low-pass filter of a two-dimensional Complex array image_complex. The cutoff frequency is given by the argument rho_0. The size of the image must be given by the argument n.

void Zonal_HP(Complex** image_complex, float rho_0, int n)

Input: Complex** image_complex, float rho_0, int n

Output: Complex** image_complex

This routine takes an in-situ zonal high-pass filter of a two-dimensional Complex array image_complex. The cutoff frequency is given by the argument rho_0. The size of the image must be given by the argument n.

void Butterworth_LP(Complex** image_complex, float rho_0, float order, int n)

Input: Complex** image_complex, float rho_0, float order, int n

Output: Complex** image_complex

void Butterworth_HP(Complex** image_complex, float rho_0, float order, int n)

Input: Complex** image_complex, float rho_0, float order, int n

Output: Complex** image_complex
This routine takes an in-situ Butterworth low-pass filter of a two-dimensional Complex array `image_complex`. The cutoff frequency is given by the argument `rho_0` and the order of the filter is given by the argument `order`. Higher order filters have steeper cutoffs. The size of the image must be given by the argument `n`.

```c
void Butterworth_HP(Complex** image_complex, float rho_0, float order, int n)
```

**Input:** `Complex** image_complex, float rho_0, float order, int n`

**Output:** `Complex** image_complex`

This routine takes an in-situ Butterworth high-pass filter of a two-dimensional Complex array `image_complex`. The cutoff frequency is given by the argument `rho_0` and the order of the filter is given by the argument `order`. Higher order filters have steeper cutoffs. The size of the image must be given by the argument `n`.

```c
void median_filter(float** image_corr, float** med_img, int median_3_or_5, int n)
```

**Input:** `float** image_corr, float** med_img, int median_3_or_5, int n`

**Output:** `float** med_img`

This function takes a two-dimensional array `image_corr`, which contains an image corrupted by noise, and performs median filtering on it. The filtered image is returned in the two-dimensional array `med_img`. A median filter of length three or five is chosen through the `median_3_or_5` argument. To choose a median filter of length three, `median_3_or_5` must be set to 3 and to choose a median filter of length 5, `median_3_or_5` must be set to 5. The size of the image must be specified through the argument `n`.

**Associated routines:**

```c
int FFT2D(Complex** image, int n, int direction)
```

**Input:** `Complex** image, int n, int direction`

**Output:** `Complex** image`

After performing a forward FFT on an image and performing the inverse transform after the appropriate filtering function is carried out, it will be necessary to convert it from type Complex to float in order to output the image. The maximum value and average value of the image is returned through the referenced values `max_val` and `avg_val`.

```c
void shift_2d_complex(Complex** array, int n)
```

**Input:** `Complex** array, int n`

**Output:** `Complex** array`

The function `shift_2d_complex()` performs the same function as `shift_2d_float()`, but only on two-dimensional Complex images. When the FFT of an image is taken, the DC component is at the corners of the resultant image when the inverse transform is displayed. Before a filter is applied, it is necessary to shift the DC component to the origin, thus the second and fourth quadrants must be
swapped and the first and third quadrants must be swapped. This is due to the DFT indexing convention, which places \((0, 0)\) element not at the center of the image, but at the upper left corner. This function should be applied after the FFT of an image is taken. The user must call this function to ensure the image is properly shifted.

```c
float min3(float a, float b, float c)
   Input: float a, float b, float c
   Output: float value

float median3(float a, float b, float c)
   Input: float a, float b, float c
   Output: float value

float median5(float a, float b, float c, float d, float e)
   Input: float a, float b, float c, float d, float e
   Output: float value
```

The function \texttt{min3()} finds and returns the minimum between three numbers. The functions \texttt{median3()} and \texttt{median5()} finds and returns the median between three values and five values, respectively. The \texttt{median3()} and \texttt{median5()} functions are called by \texttt{median_filter()}, depending on the argument \texttt{median_3_or_5}.
Image Operations

Main routines:

```c
void add_impulse_noise(float** image, float percent, int n)
void add_gaussian_noise(float** image, float snr_db, float* mean_noise_power, int n)
void scale_array(float** image, float scalar, int n)
void add_scalar(float** image, float scalar, int n)
void initialize_float_array(float** image, float value, int n)
void duplicate_float_array(float** original, float** copy, int n)
void array_superposition(float** x, float** y, float** result, float a, float b, int n)
void Complex_addition(Complex a, Complex b, Complex* result)
void Complex_subtration(Complex a, Complex b, Complex* result)
void Complex_multiplication(Complex a, Complex b, Complex* result)
void Complex_division(Complex a, Complex b, Complex* result)
```

Associated routines:

```c
long random(void)
float uniformly_dist_random_number(void)
float gaussian_dist_random_number(float std_dev)
```

These routines perform various operations on images and on Complex variables. Uniformly distributed and zero-mean Gaussian distributed random number generators are included. They are intended to simplify the programming task.

Main routines:

```c
void add_impulse_noise(float** image, float percent, int n)
    Input: float** image, float percent, int n
    Output: float** image

This function adds impulsive noise in-situ to a two-dimensional float array image. The amount of the image to be corrupted is determined by the percent argument. The size of the image must be given by the argument n.
```

```c
void add_gaussian_noise(float** image, float snr_db, float* mean_noise_power, int n)
    Input: float** image, float snr_db, float* mean_noise_power, int n
    Output: float** image

This function adds zero-mean Gaussian noise in-situ to a two-dimensional float array image such that such that the resulting SNR measured in dB is snr_db. The mean noise power is computed and returned by reference through the argument mean_noise_power. The size of the image must be given by the argument n. Note that even if all the data values are originally non-negative, the resulting array can contain some negative values.
```
void scale_array(float** image, float scalar, int n)
    Input: float** image, float scalar, int n
    Output: float** image

This routine takes a two-dimensional float array image and multiplies each element by the scalar specified by the argument scalar. The size of the image must be given by the argument n.

void add_scalar(float** image, float scalar, int n)
    Input: float** image, float scalar, int n
    Output: float** image

This routine takes a two-dimensional float array image and adds to each element the scalar specified by the argument scalar. The size of the image must be given by the argument n.

void initialize_float_array(float** image, float value, int n)
    Input: float** image, float value, int n
    Output: float** image

This function takes a two-dimensional float array image and initializes each element to the value specified by the argument value. The size of the image must be specified through the argument n.

void duplicate_float_array(float** original, float** copy, int n)
    Input: float** original, float** copy, int n
    Output: float** copy

This function takes a two-dimensional float array original and makes a copy of itself in the array copy. Both arrays must be of the same size and specified through the argument n.

void array_superposition(float** x, float** y, float** result, float a, float b, int n)
    Input: float** x, float** y, float** result, float a, float b, int n
    Output: float** result

This function takes the linear combination of two two-dimensional float arrays x and y and stores it in the array result. More precisely, \(a \cdot x[i] + b \cdot y[i] = result[i]\). The size of the arrays must be the same and specified through the argument n.

void Complex_addition(Complex a, Complex b, Complex* result)
void Complex_subtraction(Complex a, Complex b, Complex* result)
void Complex_multiplication(Complex a, Complex b, Complex* result)
void Complex_division(Complex a, Complex b, Complex* result)
    Input: Complex a, Complex b, Complex* result
    Output: Complex* result
These functions take two Complex numbers \( a \) and \( b \) and performs the specified operation in the form \( a \text{ operation } b \). The result of the operation is passed by reference through the argument \textit{result}.

**Associated routines:**

\texttt{long random(void)}

\textbf{Input:} none  
\textbf{Output:} long value

This function implements the minimal random number generator of Park and Miller. It returns a random integer of type long between zero and \((2^{31})-1 = 2,147,483,647\). The period of the random sequence is 2,147,483,647.

\texttt{float uniformly_dist_random_number(void)}

\textbf{Input:} none  
\textbf{Output:} float value

This function returns a random floating number that is uniformly distributed between 0.0 and 1.0 using the function \texttt{random()}. This function is called by \texttt{add_impulse_noise()} to generate the locations of corrupted pixels and also by \texttt{gaussian_dist_random_number()} to generate a Gaussian distributed random number.

\texttt{float gaussian_dist_random_number(float std_dev)}

\textbf{Input:} float \texttt{std\_dev}    
\textbf{Output:} float value

This function returns a random floating number that is Gaussian distributed. The standard deviation must be specified through the argument \texttt{std\_dev}. This function is called by \texttt{add\_gaussian\_noise()} to corrupted pixels.
**Image Statistics**

**Main routines:**

void histogram(unsigned char** image, int* hist, int n)
void image_statistics(float** image, float* mean, float* std_dev, int n)
float energy_complex(Complex** image, int n)
float nmse(float** original, float** reconstructed, int n)
float maximum_value(float** image, int positive_or_negative, int n)
float minimum_value(float** image, int positive_or_negative, int n)
int find_zero(float** image, int n)

These routines return specific information about a desired image.

**Main routines:**

void histogram(unsigned char** image, int* hist, int n)
  **Input:** unsigned char** image, int* hist, int n
  **Output:** int* hist

  This function takes the histogram of a two-dimensional unsigned char array **image** and the results are through array **hist**. The size of the image must be given by the argument **n**.

void image_statistics(float** image, float* mean, float* std_dev, int n)
  **Input:** float** image, float* mean, float* std_dev, int n
  **Output:** float* mean, float* std_dev

  This function finds the mean and standard deviation a two-dimensional float array **image**. The results are returned by reference through the arguments **mean** and **std_dev**. The size of the image must be given by the argument **n**.

float energy_complex(Complex** image, int n)
  **Input:** Complex** image, int n
  **Output:** float value

  This routine returns the energy (sum of squares of absolute values of all elements) in a two-dimensional Complex array **image**. The size of the image must be given by the argument **n**.

float nmse(float** original, float** reconstructed, int n)
  **Input:** float** original, float** reconstructed, int n
  **Output:** float value
This function is useful for evaluating image restoration techniques. It returns the normalized mean square error between the two-dimensional float arrays original and reconstructed as a percentage. The size of the image must be given by the argument \( n \).

\[
\text{float maximum_value(float** image, int positive_or_negative, int n)}
\]

Input: float** image, int positive_or_negative, int n 
Output: float value

This function takes a two-dimensional float array image and returns the maximum positive or negative value. The function returns the maximum positive value if the argument positive_or_negative is 1, and it returns the maximum negative value if the argument is 0. The size of the image must be specified through the argument \( n \).

\[
\text{float minimum_value(float** image, int positive_or_negative, int n)}
\]

Input: float** image, int positive_or_negative, int n 
Output: float value

This function takes a two-dimensional float array image and returns the minimum positive or negative value. The function returns the minimum positive value if the argument positive_or_negative is 1, and it returns the minimum negative value if the argument is 0. The size of the image must be specified through the argument \( n \).

\[
\text{int find_zero(float** image, int n)}
\]

Input: float** image, int n 
Output: 1 (success) or 0 (failure)

This function takes a two-dimensional float array image and determines whether or not there is a zero in the array. The size of the image must be specified through the argument \( n \). If a zero is found, then a 1 is returned, otherwise, 0 is returned. This function is called by minimum_value().
Passing by Reference:

There is often confusion over how to pass an argument by reference. In the function declaration, the function argument should be declared as a pointer to the desired type. When passing a variable, precede it by the & (ampersand) operator, which extracts the address of the variable. In the function, the variable is accessed through the pointer. The * (dereference) operator is needed to retrieve or assign data to the location which the pointer points to. By default, all arrays are passed by reference in C. This avoids copying massive structures to the argument stack. Consider this example of passing by reference:

```c
void foobar(int pass_by_value, float* one_dimensional_array, unsigned int* pass_by_reference);

void main(void)
{
    int number, return_value;
    float* vector = (float*) malloc(10 * sizeof(float));

    foobar(number, vector, &return_value);    /* Note the ampersand in front of the argument! */
    fprintf(stderr, "The value returned by reference is: %d\n", return_value);

    free(vector);
}
```

Dynamic Memory Allocation and De-allocation for Two-Dimensional Arrays:

Dynamic memory allocation and de-allocation in C is fairly straightforward in C using the `malloc()` and `free()` functions. To create an array of two dimensions in C, use the following piece of code:

```c
Complex** image_complex;

/* Allocate memory for a vector of Complex* length n. */
image_complex = (Complex**) malloc(n * sizeof(Complex*));

/* Allocate memory for n vectors of length n, resulting in a matrix of size n-by-n. */
for (i = 0; i < n; i++)
{
    image_complex[i] = (Complex*) malloc(n * sizeof(Complex));
}

/* Deallocation the memory when finished, as it is good programming practice. */
free(image_size);
```
Complex Number Representation:

A floating point, complex variable is declared as follows:

Complex z;

The real and imaginary parts of z can be easily set. For example, if you want z to have the value 3 + j2, use:

z.real = 3.0;
z.imag = 2.0;

Complex arrays are needed for Fourier transforms. For example, a statically allocated array containing N by N elements can be declared using:

Complex image_complex[N][N];

The array can also be dynamically allocated, as will be shown in the next example. The individual elements of the array can also be set using (assume here that i and j are integers, and x and y are real, floating point variables):

image_complex [i][j].real = x;
image_complex [i][j].imag = y;

Command-Line Argument Passing:

Command-line argument passing allows a convenient method for passing parameters to a program without the need for recompilation. Such parameters might include the names of files and constants required for functions. Consider the following code fragment, which takes three arguments:

/* argc gives the number of arguments, while argv contains the actual arguments. */
int main(int argc, char* argv[])
{
    /* This program has three arguments that are taken in at the command-line, two filenames and an integer. */
    char input_file[256];
    char output_file[256];
    int image_size;

    /* Absorb command line arguments. Notice that the binary executable is the first argument, that is argv[0]. */
    if ((argc > 4) || (argc < 4))
    {
        fprintf(stderr, "Usage: %s [input image] [output image] [image size]\n", argv[0]);
        exit(1);
    }
/* To copy string arguments to their appropriate locations, use strcpy(). */
strcpy(input_file, argv[1]);
strcpy(output_file, argv[2]);

/* If the argument in a number, it is necessary to call atoi() for integer arguments and atof() for float arguments. */
image_size = atoi(argv[3]);

return 1;
}

Compilation and Linking

In order for a program that uses the functions in image_lib.c to compile and function properly, it must be compiled and linked with the object file image_lib.o. Consider the following example in UNIX with the source file source_file.c (this process is automatic in Visual C++ and CodeWarrior):

1. Compile the image library source file image_lib.c and create an object file image_lib.o (-c compiler option).
   gcc –c image_lib.c

2. Compile the source file source_file.c with the desired executable name (-o compiler option). Be sure to specify the object files that need to be linked during compilation (image_lib.o) and to link the math library (-lm).
   gcc –o testprog source_file.c image_lib.o -lm
Displaying and Printing Images

ImageMagick™:
If you are on a UNIX machine, the simplest program to use to display and print your images is ImageMagick™, which is invoked at the command line by typing in `display &` (the ampersand forces the shell to launch the program in a new process). The program is straightforward in design. To access the menus, you must use left mouse button. Select File > Load… to load in the image you wish to view. The right mouse button brings up a short-cut menu which you can also load images and get information on them.

If you are using Windows 95/98/NT, Linux or the MacOS, you can download ImageMagick™ to your personal computer and convert the GRAY files into another format, such as BMP, PICT, or TIFF. You can then use a program you are familiar. ImageMagick™ can be found at the following URL:

http://www.wizards.dupont.com/cristy/ImageMagick.html

To avoid viewing or conversion problems, a GRAY file should have .gray as its file extension, a BMP file should have .bmp and so on. If you have problems or additional questions about ImageMagick™, please refer to the ImageMagick™ User’s Guide, which can be downloaded at the following URL:


Using the CONVERT program in ImageMagick™:
You may want to avoid the hassle of displaying and printing under UNIX machines. You can opt to work on UNIX, run your program on the images, and ftp your completed images to your computer. Be sure to download ImageMagick™ to your personal computer and then run the convert program so that you can view and print the image to your own printer. You may also work entirely on MacOS or Windows. The SEASnet Windows NT workstations would be a good place to work on the projects.

Converting an image from GRAY to another format is fairly straightforward using `convert`. The syntax is as follows:

```
convert[options…]file[...]
```

For example, to convert a raw GRAY image of size 768 by 512 with a 256 byte header to a Windows BMP file, use:

```
convert -size 768x512+256 image(gray image.bmp
```

If you get the following error messages:

- CONVERT: not enough pixels (filename.ext)
- CONVERT: no delegates configuration file found (delegates.mgk)
Please ignore them. Your file should have converted successfully if there were no other error
messages following these. There are many more conversion options available. Please see Chapter 8 of
the ImageMagick™ User’s Guide.
Appendix

The file `image_lib.h` contains all the function declarations used in `image_lib.c`. Please see it for useful constants and macros. The original library was written by John Villasenor (villa@icsl.ucla.edu) and was updated and rewritten to strict ANSI/ISO C/C++ compliance by Matthew Fong (mattfong@icsl.ucla.edu). The code is highly portable as a result and has been tested on Windows (Visual C++ and CodeWarrior), UNIX (gcc), and the MacOS (CodeWarrior). Please e-mail Matthew if you discover any bugs or would like to request additional functionality in the library.

Function Declarations:

```c
void read_image(unsigned char* header, unsigned char** image, char* filename, int n)
void write_image(unsigned char* header, unsigned char** image, char* filename, int n)
void write_float_values(float** image, char* filename, int n)
void generate_header(unsigned char* header, int x_dim, int y_dim)
void byte2float(unsigned char** array_in, float** array_out, int n)
void float2complex(float** array_in, Complex** array_out, int n)
void float2byte(float** array_in, unsigned char** array_out, float scale, int n)
void array_abs_val(Complex** array_in, float** array_out, float* max_val, float* avg_val, int n)
int FFT2D(Complex** image, int n, int direction)
int FFT(int dir, int m, double* x, double* y)
int DCT2D(float** image, int n, int direction)
int DCT(int dir, int m, int n, float* array)
void shift_2d_float(float** array, int n)
void shift_2d_complex(Complex** array, int n)
int Powerof2(int n, int* exponent)
void Zonal_LP(Complex** image_complex, float rho_0, int n)
void Zonal_HP(Complex** image_complex, float rho_0, int n)
void Butterworth_LP(Complex** image_complex, float rho_0, float order, int n)
void Butterworth_HP(Complex** image_complex, float rho_0, float order, int n)
void median_filter(float** image_corr, float** med_img, int median_3_or_5, int n)
float min3(float a, float b, float c)
float median3(float a, float b, float c, float d, float e)
void add_impulse_noise(float** image, float percent, int n)
void add_gaussian_noise(float** image, float snr_db, float* mean_noise_power, int n)
void scale_array(float** image, float scalar, int n)
void add_scalar(float** image, float scalar, int n)
void initialize_float_array(float** image, float value, int n)
void duplicate_float_array(float** original, float** copy, int n)
void array_superposition(float** x, float** y, float** result, float a, float b, int n)
```
void Complex_addition(Complex a, Complex b, Complex* result)
void Complex_subtraction(Complex a, Complex b, Complex* result)
void Complex_multiplication(Complex a, Complex b, Complex* result)
void Complex_division(Complex a, Complex b, Complex* result)
long random(void)
float uniformly_dist_random_number(void)
float gaussian_dist_random_number(float std_dev)
void histogram(unsigned char** image, int* hist, int n)
void image_statistics(float** image, float* mean, float* std_dev, int n)
float energy_complex(Complex** image, int n)
float nmse(float** original, float** reconstructed, int n)
float maximum_value(float** image, int positive_or_negative, int n)
float minimum_value(float** image, int positive_or_negative, int n)
int find_zero(float** image, int n)