

Help File for MILP
Medical Imaging Laboratory Program
Medical Imaging Laboratory
Department of Biomedical Engineering
The University of Iowa

Table of Contents:

I. LOADING MILP	1
II. COMPILING MILP FOR THE FIRST TIME	1
III. MAKING CHANGES TO THE MILP PROGRAM	2
IV. COMMUNICATING WITH WINDOWS APPLICATION PROGRAMMING INTERFACE (API).....	2
V. FUNCTIONALITY	3
VI. COMPILE AND LINK	3
VII. ACCESSING DATA	3
a) <i>Accessing image descriptors</i>	4
b) <i>Accessing image data</i>	4
VIII. KEY CODES AND OTHER MISCELLANEOUS INFORMATION	5
IX. DEBUGGING.....	5
APPENDIX I. MILP USES THE WINDOWS APPLICATION PROGRAMMING INTERFACE (API).....	7
APPENDIX II. AN EXAMPLE PROGRAM	8
APPENDIX III. AN EXAMPLE OF CODE TO LOAD A TIFF IMAGE.....	8
APPENDIX IV. EXAMPLE USING 2D FOURIER TRANSFORM TO REMOVE NOISE	10
APPENDIX V. A BRIEF OVERVIEW OF OBJECT-ORIENTED PROGRAMMING (OOP).....	12

I. Loading MILP

Copy the file *MILP.zip* to your directory.

Run WinZip to extract the files.

II. Compiling MILP for the first time

1. Point Microsoft's Visual Studios (VS) Version 7 (VS.NET) to the *MILP.vcproj* file. Double click (or load from the VS menu).

2. Under the *Build* menu, locate and click *Build Solution* (or hit F7). This compiles all of the components that are part of the MILP program. This also links together all of the objects into an executable image (the *MILP.exe* file).
3. Once VS loads the MILP project, you may run the program by typing F5 or by clicking the blue arrow (▶) which runs the program with the Debugger), or by typing Ctrl+F5 (or clicking on the red exclamation character (!) which runs without the Debugger).
4. The program will start, and a splash screen will appear. To remove the splash screen, click somewhere in the programs view space (or wait for a few seconds).
5. Notice the O1, O2, ..., O5 buttons on the toolbar. These are the buttons that you will be programming throughout the semester.
6. Exit the MILP program and return to VS.

III. Making changes to the MILP program

For this class, you will make all changes to the MILP program in the *Operations.cpp* file. This is the ONLY file that you will need to alter during the course. In this file you will find the location of the code to handle the program buttons O1-O5. For example you will find the following code segment that handles the event that is generated when you press the O3 button:

```
BOOL Operation3 (CMILDoc* pDoc, CImage *Image)
{
    pDoc->BeginWaitCursor();
    // TODO: Add your command handler code here

    pDoc->EndWaitCursor();
    return true;
}
```

You will add code immediately following the *//TODO* line but before the *pDoc->EndWaitCursor();* line.

IV. Communicating with Windows Application Programming Interface (API).

MILP communicates with the Windows operating system through a set of classes (see the Appendix for a definition of class) defined by Microsoft; these are called the Microsoft

Foundation Classes (MFC). An MFC-based program must handle messages that respond to the API. The MFC enable you to handle individual messages generated from pressing a toolbar button (such as O1-O5), or from a popup dialog, or other. These messages must be handled in the document's view class (in *MILView.cpp* in MILP). In MILP, code in the view class calls appropriate code in the routine that you will program (*Operations.cpp*). The Windows API does not respond to command-line programming (as does UNIX, or MATLAB). As written, the program responds to toolbar events, mouse events, and some keyboard events.

If the program is compiled in *Debug* mode, however, MILP provides a console window with which you can communicate with standard C or C++ input/output functions (see below for a more complete discussion of debugging techniques). If the program is compiled in the *Release* mode, the console window is not available.

V. Functionality

There is a minimum set of image processing functions included in the MILP program. Over the course of the semester you will add your own functions. These will be handled by pressing the O1-O5 buttons. If you wish, you may press these in sequence, so O2 does the first processing step, O3 does the next processing step, and so on.

VI. Compile and Link

Once you have made changes to the *Operations.cpp* file, compile and link the program. To do so, click on *Build | Solution*, or press F7, or click in the build icon. Note: You may wish to clean the build (this deletes all intermediate files with the .obj extension) and rebuild the project "from scratch." To do this, choose *Build | Clean Solution* and then compile and link as before.

VII. Accessing data

All information about an image (including the pixel or voxel values) is contained in a class called **CImage**. This class is defined in the files *Image.h* and *Image.cpp*. There is a standard procedure for accessing the contents (objects) in the **CImage** class. In the *Operations.cpp* file, the instance of the Image object is always called *Image*. Without the jargon, this means that the name of the data that MILP loads from an image file is always called *Image*.

The image *Image* contains the width of the image (number of columns), the height of the image (the number of rows), the number of slices (the depth or z-axis dimension), the actual pixel or voxel values (the data), variables that indicate color or gray image, etc.

a) Accessing image descriptors

The convention in the *Operations.cpp* file is that to access the image descriptors (number of rows, columns, etc.) one must use member functions. In standard C++, one should not be able to change the image descriptors directly. The reason for this is illustrated in the following example. Suppose that one of the descriptors is the image volume (the number of words required to store the image). If one changed the number of rows and columns directly (for example with a *resize* function) but forgetting to change the number of words required, then the number of required words would be incorrect. However, if the member variables (descriptors) were changed in a member function, then the function could recalculate the volume (number of words) if the size of the image were changed.

To access the integer variable *nr*, the number of rows (the image height), the code to be used in *Operations.cpp* is

```
nr=Image->GetHeight();
```

To access the integer variable *nc*, the number of columns (the image width), the code is

```
nc=Image->GetWidth();
```

To access the integer variable *ns*, the number of slices (the image depth), the code is

```
nc=Image->GetSlice();
```

To access the total number of words required to store the image (the volume), the code is

```
volume=Image->GetVolume();
```

b) Accessing image data

To access the image data is, one must remember that the pixel or voxel data are actually stored as an ordered array of numbers. The order is always row-major, that is, the data are stored as follows: first row, followed by second row, then third row, and so on. Remember that in C++, indices start at zero (0). Therefore, row numbers go from 0 to *nr-1*, and column numbers go from 0 to *nc-1*. Thus the data in row *n* and column *m* would be indexed as (*n-1*) and (*m-1*).

To access the pixel value (assuming gray image in which the pixel value is stored as one memory unit) stored at row *n* and column *m*, one would use the following code:

```
pixel_value=Image->data[n*nc+m];
```

To access data stored at row n , column m and slice p , one would use the following code:

```
offset = nr*nc*p;  
voxel_value=Image->data[n*nc+m+offset];
```

All image data (pixel or voxel values) are stored as signed integer data. The actual word size can be found from the C function *sizeof(int)*.

VIII. Key codes and Other Miscellaneous Information

There are a few key codes that have been programmed into MILP. These are the following:

- F1 – Brings up this help file (in pdf) from the class web site
- F2 – Converts a color image (in RGB format) to gray image
- F3 – Linearly adjust the color of the image between the lowest 1% and highest 99% of the pixel (voxel) values
- Down arrow or Left arrow – Display the previous slice in a volumetric image
- Up arrow or Right arrow – Display the next slice in a volumetric image
- Home – Display the first slice in a volumetric image
- End – Display the last slice in a volumetric image

MILP can call MATLAB functions. To turn this feature ‘ON’, Go to the *MIL.h* file and comment out the line `#undef USE_MATLAB_FUNCTIONS`. An example of how to call MATLAB functions from within MILP is in *MILView.cpp* under the function *OnViewHistogram()*.

IX. Debugging

Visual Studios offers various ways to help debug a program. For example, one can use TRACE macros; this is similar to `fprintf` function in C language. The output is displayed in the Output/Debug window in VS C++. These macros are ignored if the routines are compiled in Release mode, i.e., these are executed ONLY if compiled in Debug mode, AND executed with **Start** (F5), not with **Start without Debugging** (Ctrl+F5).

Examples are:

```
TRACE0 ("Use this for string text only\n");  
TRACE1 ("Use this for one argument. Volume: %d\n",Image->GetVolume());  
TRACE2 ("Use this for two arguments. Rows: %d, Cols: %d\n",
```

```

        Image->GetHeight(),Image->GetWidth());
TRACE3 ("Etc. %d %d %s\n",Image->GetBitsPerPixel(),
        Image->GetBytesPerPixel(),Image->GetFileName());

```

One can also use a popup box for debugging. This technique works when the routines are compiled in either Release mode or Debug mode. For example,

```

CString str;
str=_T("This is a test for output without arguments");
AfxMessageBox(str);
str.Format("This is a test for file %s",Image->GetFileName());
AfxMessageBox(str);
    //Strings can also be combined as :
str=_T("This is the first part\n");
str = str + "This is the second part";
AfxMessageBox(str);

```

The last method is available only to MILP (it is not part of Visual Studios) and only if the routines are compiled and built in Debug mode. When starting the program in Debug mode, a command window opens into which you can read/write (as in C language) with `canf`, `printf`, `cout`, `cin` and `cerr` functions. This simply redirects the output/input from a console app to a Windows app. For example,

```

printf ("File: %s\n",m_sResults);

```

will print to the console window the file name stored in the CString **m_sResults**.

Appendix I. MILP uses the Windows Application Programming Interface (API).

Visual C++ lets you develop a Windows API program in two stages. First, you use Visual C++'s set of tools to generate code for a program automatically; then you modify and extend the code to suit your needs. As the basis for doing this, Visual C++ uses a hierarchy of classes called the Microsoft Foundation Classes (MFC). A standard MFC-based program is structured using a Document/View concept. This architecture is the standard architecture of almost all Microsoft Windows programs and applications.

What is this Document and View architecture? Briefly:

1. What is a Document? A document is the name given to a collection of data in your application with which the user interacts. Though the word "document" seems to imply something textual in nature, a document is not limited to text. In our case, the document is actually the image data (pixels or voxels). You may not be surprised to learn that a document in your program will be defined as an object of a document class. Your document class will be derived from the class **CDocument** in the MFC library. In the MILP program, the image data is "stored" in the document class **CMILDoc** that is defined in the *MILDoc.cpp* file. MILP allows you to have more than one document open at one time (MILP uses the Multiple Document Interface or MDI).
2. What is a View? A view relates to a particular document object. A view is an object that provides a mechanism for displaying some or all of the data stored in a document. It defines how the data are to be displayed in a window, and how the user can interact with the data. The MILP program derives its view class from the MFC class **CView** (actually **CScrollView** which derives from **CView**). The view class in MILP is **CMILView**, and is defined in the *MILView.cpp* file. A view is displayed in its own window. A document object can have more than one view associated with it.

How to link a document and its view? Each view of a document object must receive a pointer to the document object. In *Operations.cpp*, the pointer is passed to the individual routines as an argument, as in the example code for handling the event generated when you press the O4 button:

```
BOOL Operation4 (CMILDoc* pDoc, CImage* Image)
```

The pointer to the document is then *pDoc*.

Appendix II. An example program

Let's write code that could appear in *Operations.cpp* that will calculate the average value of all of the pixels in an image. We will display the results in a pop-up box.

```
float sum,avg; //Set aside memory locations for sum and average
int i;        //a simple counter
int nr,nc;    //rows and columns
int ns;       // Number of slices (ns=1 for 2D image)
CString str;  //A CString is needed to output in a popup box

nr=Image->GetHeight();    //Get rows and columns
nc=Image->GetWidth();
ns=Image->GetSlices();

sum=0.0;

for (i=0; i<nr*nc; i++) { //Assume ns=1;
    sum = sum + Image->data[i]; //Get the data and sum
}
avg = sum / static_cast <float>(nr*nc);

str.Format("The average is: %f",avg);
AfxMessageBox(str);
```

Appendix III. An Example of Code to Load a Tiff Image

Let's look at a coding example to load a picture from a file (in tiff format) into the Image class. MILP can read most image file formats. If the image is not in tiff format, however, MILP must have write access to the directory from which the image is being read. Files not in tiff format can be read by selecting *Open Image* from the *File* drop-down menu.

```
CString sFileName; //This will receive the file name
CString sExtensions = "Tif Image Files (*.tif)|*.tif|All files (*.*)|*..*|"; //This
will show the possible extensions

CFileDialog
m_ldFile(TRUE,NULL,NULL,OFN_READONLY,sExtensions,NULL);
//This statement will prepare a popup file select window
```

```

        if (m_ldFile.DoModal() == IDOK) {           //Pop up the files select dialog
            sFileName = m_ldFile.GetPathName();    //The selected file
            if(TifRead(Image,sFileName)==TRUE) {
/*This function reads the tif file. Other formats are some DICOM files, and almost any
image files, but these are available only from the drop-down menus */

                pDoc->m_b2DImage=TRUE;
                // Flag to indicate 2D views desired; 3D views are available as cine (movie)
                if (pDoc->m_bFirstTime)
                //Flag to indicate file read in for the first time (there is no backup file to "undo to"
                    pDoc->m_bFirstTime=FALSE;
                //More bookkeeping - this is not the first time that files has been read in, so a backup
                image exits
                    else
                        pDoc->CopyUndoImage(); //Provides "Undo"
capabilities
                        Image->SetFileName(m_ldFile.GetFileName());
                        //stores file name in Image class
                        pDoc->m_bLoadImage=true;
                        //Indicates that an image has been loaded
                        pDoc->m_b3DImage=FALSE; //Do not display as a 3D
image
                    } else {
                        AfxMessageBox("ERROR - File read error",MB_ICONERROR);
                //Error in member function
                        return false;
                    }
                } else {
                    AfxMessageBox("ERROR - File select error",MB_ICONERROR);
                //Error in file select dialog
                        return false;
                }
                }
/*If we reach here, all is well - we have an image that you can now "work on" and
display */

                // Finished
                pDoc->SetModifiedFlag();
                //This will flag Modified at exit time
                pDoc->UpdateAllViews(NULL);
                //This actually displays the image on the screen

```

Appendix IV. Example using 2D Fourier Transform to Remove Noise

```

BOOL Operation5 (CMILDoc *pDoc, CImage *Image)
{
    // Let's save some memory locations for our variables
    int i,j,r,c;
    int volume,nc,nr;
    int nrf,ncf;
    float* Spectrum=NULL;
    double* Real=NULL;
    double* Imag=NULL;
    int Direction;
    int nrby2,ncby2;
    int size;
    float fraction;
    BOOL SquareFilter;

    //Get minimum size that is a power of 2
    size=GetPaddingSize (Image);

    //Allocate memory for Fourier arrays
    Spectrum = new float [size*size];
    Real = new double [size*size];
    Imag = new double [size*size];

    nr=size;          //size of Fourier arrays
    nc=size;
    nrby2=size/2;
    ncby2=size/2;

    Direction = 0; //Forward FFT

    if (Fourier2D (Image, Spectrum, Real, Imag, size, Direction) ==
FALSE) {
        AfxMessageBox ("Error in calculating forward Fourier
Transform",MB_ICONERROR);
        delete [] Spectrum;
        delete [] Real;
        delete [] Imag;
        return false;
    }

    //Remove the "noise"

    fraction = 0.5; // Fraction of Nyquist frequency used in filter
operation

    CInputDialog dlg;          //Creates an instance of a popup dialog
    dlg.m_sInputText="Enter desired pixel value"; //Text to be
displayed in the popup dialog

```

```

    dlg.m_fValue=fraction; //Just an initial value to display
    if (dlg.DoModal() == IDOK) { //Show the dialog, and wait
until the user enters
        fraction=dlg.m_fValue; //m_fValue is a floating point
number
    } else { //The cancel button was pressed
        delete [] Spectrum; //free up memory so others
can use it
        delete [] Real;
        delete [] Imag;
        return false; //Do nothing
    }

    nrf=fraction*nr/2; // nr/2 and nc/2 represent the
Nyquist frequency
    ncf=fraction*nc/2;

    SquareFilter=false; //Make true if square filter
desired, else circular

    ///Let's implement the filter here
    if (SquareFilter) {
        for (int c=nc-ncf;c<nc; c++) {
            for (int r=0;r<nr;r++) {
                Real[r*nc+c]=0;
                Imag[r*nc+c]=0;
            }
        }
        for (int c=0;c<ncf; c++) {
            for (int r=0;r<nr;r++) {
                Real[r*nc+c]=0;
                Imag[r*nc+c]=0;
            }
        }
        for (int r=nr-nrf;r<nr; r++) {
            for (int c=0;c<nc;c++) {
                Real[r*nc+c]=0;
                Imag[r*nc+c]=0;
            }
        }
        for (int r=0;r<nrf; r++) {
            for (int c=0;c<nc;c++) {
                Real[r*nc+c]=0;
                Imag[r*nc+c]=0;
            }
        }
    } else { //Implement as a circular filter
        for (r=0; r<nr; r++) {
            for (c=0; c<nc; c++) {
                if (sqrtf((r-nrby2)*(r-nrby2) + (c-ncby2)*(c-
ncby2))>nrf){ //Low pass
                //if (sqrtf((r-nrby2)*(r-nrby2) + (c-ncby2)*(c-
ncby2))<nrf){ //High pass
                    Real[r*nc+c]=0;

```

```

                                Imag[r*nc+c]=0;
                                }
                                }
                                }
}

memset (Spectrum,0,sizeof(float)*size*size);

Direction = 1; //Backwards

if (Fourier2D (Image, Spectrum, Real, Imag, size, Direction) ==
FALSE) {
    AfxMessageBox ("Error in calculating forward Fourier
Transform",MB_ICONERROR);
    delete [] Spectrum;
    delete [] Real;
    delete [] Imag;
    return false;
}

LinearTransform(Image,Spectrum);
pDoc->UpdateAllViews(NULL); //This actually displays the
image on the screen
AfxMessageBox("Image after filtering");

delete [] Spectrum;
delete [] Real;
delete [] Imag;

pDoc->SetModifiedFlag(); //This will flag
Modified at exit time
pDoc->UpdateAllViews(NULL); //This actually displays the
image on the screen

return true;
}

```

Appendix V. A brief overview of Object-Oriented Programming (OOP)

Note: Check out <http://www.cplusplus.com/main.html> for brief tutorials on C++.

As in the case with C, the data types in C++ are divided into two groups: **native** and **user defined**. The **native** types are those defined in the language itself. Examples of these are *int*, *long int*, *float*, *double*, etc. The **user defined** types are defined by the user. There is a major difference between the user defined types in C and C++. In general, C++ provides much better support for user defined data types.

There are a couple of definitions that we all should know.

A **class** is a user defined data type. In MILP, we have defined a class called **CImage** that contains all of the information about an image. The information includes, but is not limited to, the number of rows, the number of columns, the data, etc. Also in the class are functions to changes these variables, functions to interrogate these variables, functions to calculate volume, and other derived variables.

A **class interface** tells the compiler what facilities or capabilities the class provides. This interface is usually found in a header file, which by convention has the extension *.h*. For our MILP example, the interface for the **CImage** class is *Image.h*.

A **class implementation** tells the compiler how to implement the facilities or capabilities defined in the class interface. This is usually found in a source file, which has the extension *.cpp*. For the MILP example **CImage** class, the implementation file is *Image.cpp*.

An **object** is a variable of a class type. Its behavior is defined by the code that implements the class. For example, the object *volimg* is a variable in MILP of the class type **CImage**. The object *volimg* is an instance (a particular example) of the class.

A **member function** is a function that is part of the definition of the class. For example, the member function *GetWidth()* returns the width of the object *volimg*.

A **member variable** is a variable that is part of the definition of a class. For example, the member variable *m_nc* contains the number of columns in the object *volimg*.

Object-Oriented Programming (OOP) is the organization of programs as a collection of objects, rather than as collections of functions operating on variables of native data types. Most real-world applications are actually implemented in an object-oriented programming language such as C⁺⁺.

Encapsulation is the concept of hiding the details of a class inside the implementation of that class rather than exposing them in the interface. This is one of the primary organizing principles that characterize OOP. The MILP program does not take full advantage of encapsulation, but a large part of the operations on an image is conducted within the class.

A **concrete data type** is a class whose objects behave like variables of native data types.

A **constructor** is a member function that creates new variables of the class type. All constructors have the same name as the class for which they are constructors.

A **default constructor** is a constructor that is used when no initial values is specified for an object. Since it has no values, it has no arguments. The default constructor for the image class **CImage** is found in *Image.cpp* to be (copied only in part)

```
CImage::CImage()  
  
    {  
  
        ...  
  
    }
```

A **copy constructor** makes a new object with the same contents as an existing object of the same type. A copy constructor essentially makes an exact copy of an object.

An **assignment operator** is a member function that sets a pre-existing object the same value as another object of the same type.

A **destructor** is a member function that cleans up when an object expires; for a local object, this occurs at the end of the function where that object is defined.