IPTK – Image Processing Toolkit

S. Murali Krishna, Bh. M. Bharadwaj, P. Krishna Mohan, T. Suresh Babu
Chakravarthy Bhagvati

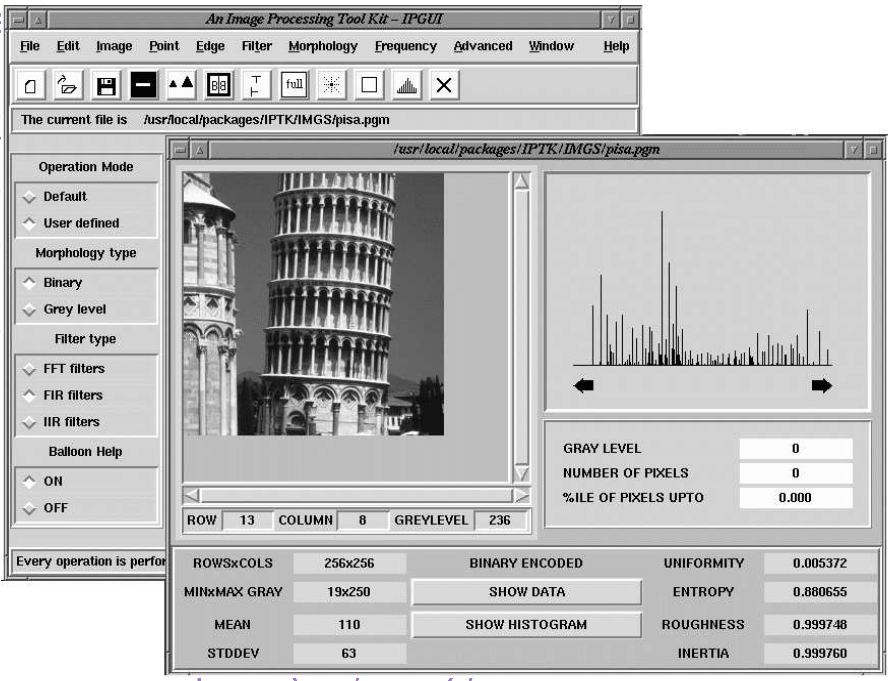
```
#include "hipl.h"
int ma
{
    i
    T
    D
    i                                        Wspc;

    i
    o
    I
    H
    H
    H
    H

    H
    HIPL_StoreResult(argv[2], ot);

    HIPL_FreeMem(it);
    HIPL_FreeMem(ot);
    HIPL_Closeall(it);
    HIPL_Closeall(ot);

    exit(0);
}
```



The current file is /usr/local/packages/IPTK/IMGS/pisa.pgm

Operation Mode
- Default
- User defined

Morphology type
- Binary
- Grey level

Filter type
- FFT filters
- FIR filters
- IIR filters

Balloon Help
- ON
- OFF

Every operation is perfor

ROW	13	COLUMN	8	GREYLEVEL	236

GRAY LEVEL	0
NUMBER OF PIXELS	0
%ILE OF PIXELS UPTO	0.000

ROWSxCOLS	256x256	BINARY ENCODED	UNIFORMITY	0.005372
MINxMAX GRAY	19x250	SHOW DATA	ENTROPY	0.880655
MEAN	110	SHOW HISTOGRAM	ROUGHNESS	0.999748
STDDEV	63		INERTIA	0.999760

Dept. of Computer and Information Sciences
University of Hyderabad

CHAPTER 1 OVERVIEW

IPTK is a powerful image processing environment developed by students of the Department of Computer and Information Sciences at University of Hyderabad. **IPTK** works at three levels. There is a *GUI* that allows lay users to perform simple image processing operations at the click of a button. The second level is a *Tcl/Tk* interpreter with image processing extensions. It provides a fast prototyping environment that combines the powerful scripting facilities of Tcl/Tk with the variety of image processing functions of **IPTK**. The third level is a library of C-callable functions for complete control over writing new applications. This manual primarily describes **IPTK** library and programming with the functions available therein.

1.1 Font Conventions

We use the following fonts to indicate different types of information.

- Words that a user must type *verbatim*, i.e., as given in the manual are given in typewriter font.

  ```
  iptk -f ipgui.tcl &
  ```

- Words/phrases that a user must *substitute* with an appropriate word are given in an italicized font.

  `edge_detect` *`input-image output-image`*

  In the above example, the user must type the first word as it is but substitute the name of appropriate images for the remaining two.

- Words/phrases/sentences that are output by **IPTK** are given in helvetica font.

  Error in allocating memory.

- Warnings and other important instructions or idiosyncracies of **IPTK** are given in a boxed paragraph using bold text

  You must always check the return values from initialise function

1.2 Package Contents and System Requirements

IPTK is available on a single floppy disk that contains source files (.c files), makefile to create the executable version, shell-scripts to create the library and a set of sample images. It also contains this manual in the file `iptk-ref.pdf`. Ensure that the floppy contains the following two files:

1. IPTK.tgz: a gzipped tar containing **IPTK**
2. iptk-ref.pdf.gz: documentation contained in this manual

IPTK requires about 6MB of disk space when unzipped. System requirements are given below.

- Linux/Unix operating system. The C-callable library works even on Windows with a VC++ compiler (tested with Win9x, any volunteers for others?) but the GUI and the Tcl/TK interpreter extensions do not work
- At least 32MB RAM recommended although **IPTK** is designed to work with even 512KB RAM. The more memory there is, the faster **IPTK** runs
- Tcl/TK (at least Version 8.0, although tested with Version 4.0)
- gcc compiler (any ANSI C compiler will do)
- Monitor capable of 800×600 resolution with at least 8-bit depth (i.e., capacity to display 256 colours or grayscales)

CHAPTER 2 INSTALLATION

2.1 Instructions for Linux OS

Make sure you have the file `IPTK.tgz` with you. It is a compressed tar file that contains the source code and several test images. The size of the gzipped file is approximately 520KB and fits easily on a floppy.

Follow the instructions below after getting the above two files from me.

1. Create the directory `/usr/local/packages/IPTK` on your system. You need to be **root** to do this
2. Copy IPTK.tgz into `/usr/local/packages/IPTK` directory
3. Change to `/usr/local/packages/IPTK` directory
4. Type `tar xvzf IPTK.tgz` to extract IPTK files
5. Type `make` to install IPTK. Normally this should create `iptk` executable correctly. If there are any problems, they can be fixed by editing the file `makefile`
6. The following steps make running **IPTK**more convenient
   - (a) Change to `/usr/local/bin` directory
   - (b) Save the following lines into a file called `iptk`

     ```
     #!/bin/sh
     /usr/local/packages/IPTK/iptk -f \
               /usr/local/packages/IPTK/ipgui.tcl
     ```
   - (c) Make the script executable by typing `chmod 755 iptk`
7. Change back to `/usr/local/packages/IPTK` directory.
8. Type `./shellfile.sh` to create the **IPTK**library `libHIPL.a`
9. Do the following to put the library and header files in their correct directories for compilers to use them later

(a) Change to `/usr/lib` directory and create a link to **IPTK**library by typing

```
ln -s /usr/local/packages/IPTK/libHIPL.a .
```

(b) Change to `/usr/include` directory and create a link to **IPTK**header file `HIPL.h` by typing

```
ln -s /usr/local/packages/IPTK/HIPL.h .
```

10. Finally, set the environment variable `IPTKHOME` by typing

```
export IPTKHOME=/usr/local/packages/IPTK
```

This may be done once and for all by adding the line in your `.bashrc` file

You are now done. Change to your home directory and type `iptk`. It should bring up the **IPTK**interface. To test the installation of **IPTK**library, copy the file `sample.c` into your directory. Compile it by typing

```
gcc -o sample sample.c -lHIPL -lm
```

You should get the executable file `sample` if everything is properly installed. If there are errors, please verify that you followed all the steps correctly.

You can run `sample` on any of the test images by typing, for example,

```
./sample /usr/local/packages/IPTK/IMGS/pisa.pgm pisa-out.pgm
```

(`pisa.pgm` is one of the sample images.) The program will create a new image "pisa-out.pgm" in your directory. Take a look at it.

CHAPTER

3 PROGRAMMING WITH HIPL LIBRARY

The library that comes with the **IPTK** package is found with the file name *libHIPL.a* in the installation directory. There must be a link to this file from the `/usr/lib` directory. The **IPTK** library will be called *HIPL* library from now on.

Ensure that /usr/lib/libHIPL.a is a symbolic link to the file libHIPL.a in IPTK installation directory. Otherwise, it will not be possible to compile any of the image processing programs.

HIPL library comprises functions at three distinct levels.

1. **High-Level Operations**

   Programming at this level uses image processing functions provided by HIPL library. There is no need to access individual pixels in the image. Examples of functions at this level include `HIPL_Sobel` for performing edge detection using Sobel operator, `HIPL_Connect` for computing connected components, `HIPL_FFTlp` for applying a frequency domain low-pass filter, etc.

   This chapter deals with high-level operations only

2. **Pixel-Level Operations**

   Programming at this level is normally needed to create new image processing operations or for manipulating small regions in images. Users need to access individual pixels and therefore know about the internal structure of the *HIPL_IMAGE* data structure provided by the library. Examples of functions at this level include `HIPL_Getpart` for reading an image, `HIPL_Putpart` for writing an image, etc.

   The most often used level of programming is pixel-level operations. Developing any new image processing operation requires programming at this level. These operations are explained in the next chapter.

3. **Core Operations**

   Programming at this level is not needed for anyone except those concerned with extending **IPTK** system to handle new image formats or modifying the basic data structures used in the system. Programmers not only need to know the data structures used by **IPTK** but also must have a knowledge of image formats, underlying memory organization used by **IPTK** and internal representation of pixel values. An example function at this level is `HIPL_Gettype` for finding the format of an image. Core-level programming is not described in this booklet.

Core operations are NOT NEEDED TO DO ANY IMAGE PROCESSING OPERATION. It is, therefore to be attempted only with a THOROUGH AND COMPLETE understanding of IPTK. Any mistakes at this level usually cause entire IPTK package to fail.

Compiling HIPL Programs

Any **IPTK** program needs linking the library `libHIPL.a`. The general way to compile (in Linux environment) is by typing the following command:

```
gcc <source file> -lHIPL -lm -o <executable file
                           name>
```

3.1 High-Level Programming

All **IPTK** programs contain four distinct parts — image related declarations, initialization section, processing, and finally closing section. These sections are described using the example program, `sample.c` (see Figure 3.1, which comes with the installation package.

Image Declarations

HIPL library provides `HIPL_IMAGE` data structure to handle images. There is also an associated data type called `HIPL_DATA` that is used to pass information efficiently from one image to another. These are the only two new data structures needed for programming at a high-level. All other standard data types provided by the 'C' language and other user-defined structures may be used as in a regular program written in 'C.'

```
 1    #include <stdio.h>
 2    #include <malloc.h>
 3    #include <math.h>
 4    #include <hipl.h>
 5    #include <err_hipl.h>

 6    #define SIGMA 2

 7    int main(int argc, char *argv[]) {
 8         int i;
 9         HIPL_IMAGE *it, *ot;
10         HIPL_DATA *ID;
11         extern int HIPL_ERRNO;

12         it=(HIPL_IMAGE *)malloc(sizeof(HIPL_IMAGE));
13         ot=(HIPL_IMAGE *)malloc(sizeof(HIPL_IMAGE));
14         ID=(HIPL_DATA *)malloc(sizeof(HIPL_DATA));

15         HIPL_Initialise(it, argv[1], 0, 0, ID);
16         HIPL_AllocateMem(it);
17         HIPL_Initialise(ot, NULL, 1, 0, ID);
18         HIPL_AllocateMem(ot);

19         HIPL_Gaussavg(it, ot, SIGMA);
20         HIPL_StoreResult(argv[2], ot);

21         HIPL_FreeMem(it);
22         HIPL_FreeMem(ot);
23         HIPL_Closeall(it);
24         HIPL_Closeall(ot);

25        exit(0);
26    }
```

Figure 3.1: An example high-level program using HIPL Library

In Figure 3.1, Lines 3–5 show the header files necessary for HIPL programs. Lines 1–2 and 6–8 are standard 'C' language. Lines 9–11 show the image-related declarations. In Line 9, two variables `it` and `ot` are declared as images. The variable `ID` is declared as `HIPL_DATA`. These two sets of declarations are common to **all** HIPL programs and vary only in the specific names and numbers of variables.

The header files `err_hipl.h` and `hipl.h` must be included for HIPL library to be accessible. As almost all HIPL library functions include mathematical operations, the header file, `math.h` must also be included.

Initialization Section

The declared image and `HIPL_DATA` variables must be initialized prior to use. It is common to declare image variables as pointers to the structure `HIPL_IMAGE` and therefore memory must be allocated to them. `HIPL_DATA` variables may or may not be pointers. In the sample program of Figure 3.1, `ID` is declared as a pointer and memory should be allocated for it.

Lines 12–18 cover the initialization section. Lines 12–14 allocate memory for the pointer variables. Line 15 calls the function `HIPL_Initialise` to initialize the image data structure for reading the image found in the file specified as the first argument `argv[1]` to the program. Line 16 allocates the memory for the image. Line 17 initializes the variable `ot` for an image structure to be used as output. As the output image is created by the program, its dimensions and other parameters such as type, are unknown. The normal approach in HIPL programming is to create an output image structure identical to the input image. This is achieved by passing `HIPL_DATA` variable `ID` that is initialized when initializing the variable `it` for input. Line 18 allocates memory for the output image based on the information found in `ID`.

The parameters for an output image can also be filled in manually by setting the image parameters using a lower-level **IPTK** function. This is not normally used because it requires knowledge of image formats and `HIPL_IMAGE` data structure details.

Image Processing Section

Over 40 image processing functions are provided by **IPTK** library. These functions normally take one image structure as input and another as output. Other parameters are sometimes necessary such as a *threshold* value if performing a threshold operation. No information about the implementation of `HIPL_IMAGE` structure is necessary.

The example program in Figure 3.1 performs *Gaussian Smoothing* operation. The operation is shown in Line 19 and smooths the input image `it` with the result placed in output image `ot`. Gaussian smoothing requires the width of the kernel, which is given by the parameter `SIGMA`. Line 20 stores the output image structure in the file specified as the second command-line argument `argv[2]` to the program.

Closing Section

The closing section is almost always the same for **all IPTK** programs. Lines 21–25 illustrate the general set of functions that release all the allocated memory and free the data structures and intermediate files created during the execution of the program.

Compiling and Executing the Sample Program

The sample program in Figure 3.1 is compiled by typing

```
gcc sample.c -lHIPL -lm -o sample
```

The resulting executable `sample` is run on the input image file `pisa.pgm` (shown in Figure 3.2(a)) to produce the output shown in Figure 3.2(b). The program is run with the following command.

```
sample pisa.pgm pisa-gauss.pgm
```

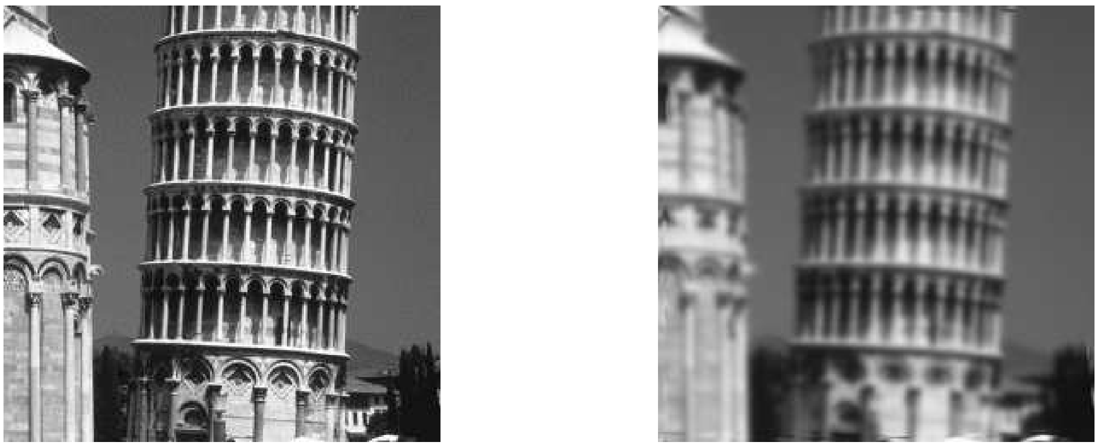


Figure 3.2: (a) Pisa image (file: pisa.pgm), (b) Result of Gaussian Smoothing with a kernel of $\sigma = 2$ (file pisa-gauss.pgm)

CHAPTER

4 PROGRAMMING AT PIXEL LEVEL

Programming at pixel level is the most common use of HIPL library. Hence it is the most important. The main structure of the program remains the same as that explained in the previous chapter. The contents of *Image Processing* section are no longer functions available in HIPL library but contain code to access individual pixels. As the other sections remain the same, they are not described in this chapter.

Image Processing Section

Image processing section contains code that manipulates individual pixels and it is necessary to know how **IPTK** handles images. The `HIPL_IMAGE` structure stores meta-information about the image such as its dimensions and type along with the actual gray-level information.

Images can be quite large – for example, an A4 document scanned at 300 dpi is nearly 2500 $\times$ 3600 pixels in size. It is therefore difficult to declare large enough arrays. It is inefficient to use a linked list for such large amounts of data. **IPTK** solves the problem by using an array whose size is dependent on the system memory. An image smaller in size than the array is read *all at once* and data is directly accessible from the array.

Larger images are read in parts whose sizes are equal to the size of the array used by **IPTK**. Several problems arise when images are handled in parts, especially when dealing with pixels at the boundaries between the parts. Two functions, `HIPL_Getpart` and `HIPL_Putpart`, are provided in HIPL library for handling all input/output issues. These two functions take care of all boundary effects and ensure that the programmer feels that the entire image is in memory.

Image processing code is mainly organized in a `HIPL_Getpart – HIPL_Putpart` loop. The sample program shown in Figure 4.3 illustrates the use of these two functions. The program checks the value of each pixel and passes to the output only those values in the range 160 – 240. If a pixel value lies outside the range, it is made 0. Such an operation is called *range slicing* and is often used to extract objects of interest.

Lines 1–14 are virtually identical to the program shown in Figure 3.1 and represent the *image declarations* and *initialization* sections. The only difference is that the variable `ID` is no longer a pointer.

Lines 15–32 illustrate pixel-level programming and are absent in the earlier program. `HIPL_Getpart` and `HIPL_Putpart` form the control for a `do...while` loop. `HIPL_Getpart` in Line 16 reads either the complete image or a part of the image (see above) and stores the pixel values in the `HIPL_IMAGE` data structure. The values are found in the array `iarr` present in the `HIPL_IMAGE` data structure. `HIPL_Getpart` returns the number of rows **read** from the image. It returns -1 in case of any error. The error-check is performed in Lines 16–18.

ALWAYS use `HIPL_Getpart` and `HIPL_Putpart` when processing images EVEN IF YOU KNOW THAT THE IMAGES ARE SMALL IN SIZE. If the images are small, there is no loss in efficiency compared to directly using an array for storing pixel values. `HIPL_Getpart` and `HIPL_Putpart` do exactly the same in such cases!

The pixel values are read in a double loop – one for the row dimension and the other for column dimension – and processed appropriately. Note that the row index `i` in Line 20 goes from 0 to `max`, the number of rows read by `HIPL_Getpart` and not `it->img.ROWS`, which is the actual number of rows in the image. The column index `j` in Line 21 follows the standard pattern of looping from 0 to `it->img.COLS`, the actual number of columns in the image. This is due to the fact that `HIPL_Getpart` **always** splits the images row-wise (see [?] for details on the functioning of `HIPL_Getpart`).

Lines 22–25 perform range slicing. Lines 26–29 write the output using `HIPL_Put part` along with error-checking. `HIPL_Putpart` returns the number of rows that remain unwritten and its return value is used to terminate the `do...while` loop (Line 30). Line 31 calls `HIPL_Refresh` function to reset all the parameters inside `HIPL_IMAGE` data structure to default values, thus making it ready to be processed by `HIPL_Getpart` or `HIPL_Putpart` again. Lines 32–38 are identical to the *Closing section* of the code in Figure 3.1.

A common programming error is to forget calling `HIPL_Refresh` after running `HIPL_Getpart` and `HIPL_Putpart`.

```c
#include <stdio.h>
#include <math.h>
#include <hipl.h>
#include <err_hipl.h>

int main(int argc, char *argv[]) {
  int i, j;
  HIPL_IMAGE *it, *ot;
  HIPL_DATA ID;

  it=(HIPL_IMAGE *)malloc(sizeof(HIPL_IMAGE));
  ot=(HIPL_IMAGE *)malloc(sizeof(HIPL_IMAGE));
  HIPL_Initialise(it, argv[1], 0, 0, &ID);
  HIPL_AllocateMem(it);
  HIPL_Initialise(ot, NULL, 1, 0, &ID);
  HIPL_AllocateMem(ot);

  do {
      if ((max = HIPL_Getpart(it)) < 0) {
         fprintf(stderr, "Error in reading image\n");
         exit(1);
      }

      for (i=0; i<max; i++)
         for (j=0; j<it->img.COLS; j++)
            if ((it->iarr[i][j]>160) && (it->iarr[i][j]<=240))
               ot->iarr[i][j] = it->iarr[i][j];
            else
              ot->iarr[i][j] = 0;

      if ((i = HIPL_Putpart(ot)) < 0) {
         fprintf(stderr, "Error in writing image\n");
         exit(1);
      }

  } while (ot->P.nleft > 0);

  HIPL_Refresh(ot, 0);
  HIPL_StoreResult(argv[2], ot);
  HIPL_FreeMem(it);
  HIPL_FreeMem(ot);
  HIPL_Closeall(it);
  HIPL_Closeall(ot);
  exit(0);
}
```

Figure 4.3: Sample program showing pixel-level operations

The result of compiling and running the program in Figure 4.3 is shown in Figure 4.4.

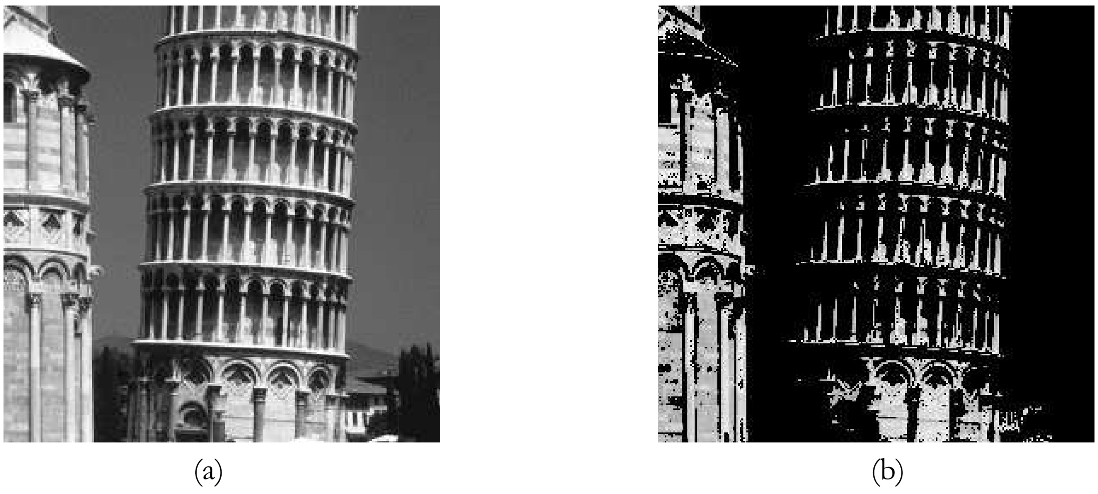


Figure 4.4: (a) Pisa image. (b) Result of range slicing between 160 and 240.

Using Mask-Based Operations

IPTK splits large images into smaller *chunks* that are processed and written into temporary files. Mask operations become tricky because of such splitting. Normally a mask of size $K \times L$, i.e., with K rows and L columns applied to a pixel $P(i, j)$ requires pixels from row $(i - \lfloor \frac{K}{2} \rfloor)$ to $(i + \lfloor \frac{K}{2} \rfloor)$ to update the value of $P(i, j)$. This leads to a problem at the boundary of the image chunk returned by `HIPL_Getpart`. To overcome this problem, mask size must be initialized using `HIPL_Maskinit` function. The prototype of this function is

```
HIPL_Maskinit(HIPL_IMAGE *it, int rows, int cols,
                    int orgx, int orgy)
```

The number of rows and columns in the mask, and the location of the origin are the parameters. For example,

```
HIPL_Maskinit(it, 3, 3, 1, 1)
```

initializes a 3×3 mask with the origin at the center of the mask.

Remember always to call `HIPL_Maskinit` before using any mask-based operation.

CHAPTER 5 LIST OF FUNCTIONS

5.1 Core Input/Output Functions

These functions are found in the file `pimg_core.c`. These functions call several functions defined in the file `pimg_pgm.c` which are not needed by IPTKusers. These latter functions are only for use by those modifying or adding to the core input/output library.

1. `int HIPL_Initialise(HIPL_IMAGE *T,char *name,short io,`
   `                    short arr_type,HIPL_DATA *ID)`

   This function initializes the image data structure. Name should refer to an existing PGM or PPM image, and memory should already be allocated for HIPL_IMAGE and HIPL_DATA. This function should be called before performing any other HIPL functions.

2. `int HIPL_AllocateMem(HIPL_IMAGE *T)`

   This function allocates memory for image data structure that is already initialized. This function should also be called before performing any image processing function.

3. `int HIPL_Maskinit(HIPL_IMAGE *it,int rows, int cols,`
   `                  int orgx,int orgy)`

   This function initializes mask parameters for many spatial and morphological operations.

4. `int HIPL_Getpart(HIPL_IMAGE *T)`

   This function reads image data into memory. The amount of data read depends on the available memory. If the image size is smaller than the memory parameter set with `HIPL_Setmemory()` function, then the entire image is read into memory. If the size of the image is larger, then a smaller number of rows are read into memory. It returns the number of rows read in.

   This function is used only when adding new image processing functions to the library. If you use only predefined functions, you don't need to see this function or `HIPL_Putpart()` in your code!

5. `int HIPL_Putpart(HIPL_IMAGE *T)`

   This function is the dual to `HIPL_Getpart()`. It writes out image data into a temporary file. The same conditions that apply to `HIPL_Getpart()` are valid for this function too.

   This function is used only when adding new image processing functions to the library. If you use only predefined functions, you don't need to see this function or `HIPL_Getpart()` in your code!

6. `int HIPL_StoreResult(char *name,HIPL_IMAGE *T, int type)`

   This function writes out image data to a specified output file. The data may be written out in either PPM or PGM format as specified by `type` parameter. If the input image is PPM and PGM is specified as the output format, then only the RED component is stored. If the original is PGM and the output format is specified as PPM, then the image is displayed as a RED image.

7. `int HIPL_Closeall(HIPL_IMAGE *T)`

   This function closes all intermediate and image files opened for reading or writing.

8. `int HIPL_FreeMem(HIPL_IMAGE *T)`

   This function frees the memory allotted for images using `HIPL_AllocateMem` function.

9. `int HIPL_Refresh(HIPL_IMAGE *T,int Flag)`

   This function resets the various parameters to their initial values. Typically, this is called after either `HIPL_Getpart()` or `HIPL_Putpart()` functions.

   This function is needed only if you are adding a new image processing function to the library.

10.
```
int HIPL_ReadImage(HIPL_IMAGE *T, char *name, short io,
                   short arr_type, HIPL_DATA *ID)
```

    This function reads a PPM or a PGM image. It combines the functionality of `HIPL_Initialise()` and `HIPL_AllocateMem()` functions.

11.
```
int HIPL_SpecifyOutParams(HIPL_DATA *ID, int NRows,
                          int NCols, int Type)
```

    This function specifies the parameters for initialising and allocating memory to an output image.

12. `void HIPL_Error(char *msg)`

    This function outputs an error message to `stderr`.

13. `void HIPL_Setmemory(int Memory)`

    This function sets the size of memory blocks that will be allocated by `HIPL_AllocateMem()` function. This block will be used by `HIPL_Getpart()` and `HIPL_Putpart()` functions to process large images.

The following functions found in the file `pimg_pgm.c` are called by the functions listed above. However, these functions need very rarely be called by the programmers or users of **IPTK**. They will not be described here but anyone intested may refer to IPTK Masters' thesis[**?**].

1. `HIPL_ReadImagePGM(FILE *fp1,HIPL_IMAGE *t)`
2. `HIPL_GetImageHeaderPGM(FILE *fp, HIPL_IMAGE *t)`
3. `HIPL_GetImageTypePGM(HIPL_IMAGE *t)`
4. `HIPL_GetImageRowsPGM(HIPL_IMAGE *t)`
5. `HIPL_GetImageColsPGM(HIPL_IMAGE *t)`
6. `HIPL_WriteImagePGM(FILE *fp, HIPL_IMAGE *t)`
7. `HIPL_PutImageHeaderPGM(FILE *fp, HIPL_IMAGE *t)`
8. `HIPL_GetImageHeaderPPM(FILE *fp, HIPL_IMAGE *t)`
9. `HIPL_ReadImagePPM(FILE *fp1,HIPL_IMAGE *t)`
10. `HIPL_WriteImagePPM(FILE *fp, HIPL_IMAGE *t)`

5.2 Point Operation Functions

1. `int HIPL_Negative(HIPL_IMAGE *T1,HIPL_IMAGE *T2)`

   This function performs digital negative operation on the input image $T1$ and stores the result in $T2$. Returns 0 upon success and a positive value in case of errors.

2.
   ```
   int HIPL_Differ(HIPL_IMAGE *it1,HIPL_IMAGE *it2,
                   HIPL_IMAGE *ot)
   ```

   This function outputs the differences in pixel values between two input images `it1` and `it2` as the output image `ot`. The two input images **must have the same dimensions**.

3.
   ```
   int HIPL_SIMax(HIPL_IMAGE *it1,HIPL_IMAGE *it2,
                  HIPL_IMAGE *ot)
   ```

   This function superimposes the input image `it2` on image `it1`. The pixel value in the output image `ot` is the maximum of the two corresponding pixels from the input images.

4. `int HIPL_SIAvg(HIPL_IMAGE *it1,HIPL_IMAGE *it2, HIPL_IMAGE *ot)`

   This function superimposes the input image `it2` on image `it1`. The pixel value in the output image `ot` is the average of the two corresponding pixels from the input images.

5. `int HIPL_HFlip(HIPL_IMAGE *it,HIPL_IMAGE *ot)`

   This function outputs a mirror-reversed image (`ot`) of the original (`it`).

6. `int HIPL_Range(HIPL_IMAGE *it,HIPL_IMAGE *ot,int lo,int hi)`

   This function performs range slicing on the input image `it` and places the result in the output image `ot`. The output range is `lo` $< g <$ `hi`. **Range** $g_i - g_{i+1}$ **is empty range.**

7. `int HIPL_Transpose(HIPL_IMAGE *it,HIPL_IMAGE *ot)`

   This function transposes the rows and columns of the input image `it` and places the result in `ot`. The effect is that of rotation by 90^o in clockwise direction. This function is defined only on **square images, i.e, number of rows is equal to number of columns**.

8. `int HIPL_Thresh(HIPL_IMAGE *it,HIPL_IMAGE *ot,int grey)`

   This function converts a grayscale image into a binary image by thresholding the pixel intensities. All pixels with intensity $\geq$ `grey` are set to *white* while the others are all set to *black*.

9. `int HIPL_Str(HIPL_IMAGE *it,HIPL_IMAGE *ot, long int hist[257],int gmax,int gmin)`

   This function performs histogram stretching. The input range is determined automatically from the image and the output range is given by `gmin - gmax`. Input image is given by `it`, and the histogram stretched image is in `ot`. The array `hist[257]`, containing the image gray level histogram, must be calculated prior to calling this function by using `HIPL_FindHistogram` function.

10. `int HIPL_Equalise( HIPL_IMAGE *it,HIPL_IMAGE *ot, long int hist[257])`

    This function performs histogram equalization. The input image is given by `it` and the output image by `ot`. The array `hist[257]`, containing the image histogram, must be calculated prior to calling this function by using `HIPL_FindHistogram` function.

11. `int HIPL_Lstretch(HIPL_IMAGE *it,HIPL_IMAGE *ot)`

    This function performs logarithmic stretching of image histogram. The input image is given by `it` and the resulting image is `ot`.

12. `int HIPL_ImgData(HIPL_IMAGE *it,long int hist[257],`
    `                 HIPL_DATA *ID)`

    This function computes several statistical and texture measures on the given input image `it`. The texture and statistical features are stored in the variable `ID`.

13. `int HIPL_FindHistogram(HIPL_IMAGE *it,`
    `                       long int hist[257])`

    This function computes the gray level histogram for the input image `it`.

14. `int HIPL_RangeStretch(HIPL_IMAGE *it,`
    `                      HIPL_IMAGE *output,`
    `                      long int hist[257],int iter)`

    This function performs *incremental* gray level stretching on the input image `it`. The parameter `iter` specifies how many times the incremental stretching is done on the input. As `iter` is increased, the image becomes a binary image. The array `hist[257]` should be initialized prior to calling this function.

5.3 Spatial Operations

1. `int HIPL_Maskimg(HIPL_IMAGE *it,HIPL_IMAGE *ot,`
   `                 int **Mask,double factor)`

   This function convolves the input image `it` with a generic mask given by `Mask` and places the result in `ot`. The parameter `factor` is the divisor of the result of convolution. For example, a 3×3 mask containing all 1s with a `factor` of 9 is the simple average mask.

2. `int HIPL_Fmaskimg(HIPL_IMAGE *it,HIPL_IMAGE *ot,`
   `                  double **Mask,double factor)`

   This function is identical to `HIPL_Maskimg` function described above except that the mask values are *double precision* numbers and not integers.