The IRAF Image I/O Interface
Design Strategies
Status and Plans

Doug Tody
November 1983

NOTE: Outdated, information only.

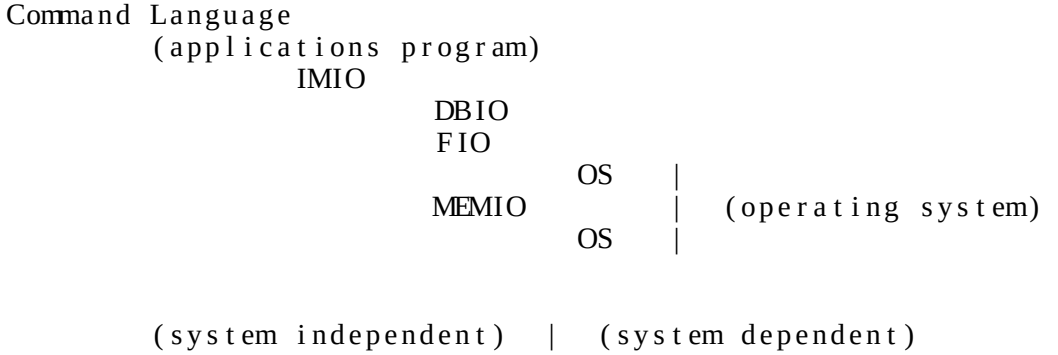
1. Introduction

Bulk data arrays are accessed in IRAF SPP programs via the Image I/O (IMIO) interface. IMIO is used to create, read, and write IRAF **imagefiles**. The term **image** refers to data arrays of one, two, or more dimensions. Each "imagefile" actually consists of two files: the **header file** and the **pixel storage file**. Seven disk datatypes are currently supported.

The IMIO calling sequences are summarized in the *Programmer's Crib Sheet*. There is as yet no Reference Manual or User's Guide for the package. Our intention in this document is merely to introduce IMIO, to summarize its capabilities, and note what is planned for the future.

2. Structure

The basic structure of an applications program which uses IMIO is shown below. In the current implementation of IMIO the image header is a simple binary structure, but this will change when DBIO (the database interface) is implemented. The pixel storage file is accessed via FIO (the IRAF File I/O interface) which permits arbitrarily large buffers and double or multiple buffered i/o. All buffers are dynamically allocated and deallocated using the facilities provided by the MEMIO interface.



3. Summary of What is Provided by the Current Interface

The IMIO interface code is mostly concerned with pixel buffer allocation and manipulation, and with mapping requests to read and write image sections into file i/o calls. FIO handles all low level i/o. The efficiency of FIO for sequential image access stems from the fact that the FIO buffers may be made as large as desired transparently to the outside world (i.e., IMIO), the number of FIO buffers is variable, and full read-ahead and write-behind are implemented (provided the OS provides asynchronous i/o facilities).

IMIO currently provides the following functions/features:

(1) 7 disk datatypes (ushort, silrdx).

(2) 6 in-core datatypes (the standard silrdx).

(3) Images of up to 7 dimensions are currently supported. The maximum dimensionality is a sysgen parameter.

(4) Fully automatic multidimensional buffer allocation, resizing, and deallocation. There is no fixed limit on the size of a buffer (a subraster may actually exceed the size of the image if boundary extension is employed). The size of an image is limited only by the resources of the machine.

(5) An arbitrary number of input buffers (default 1) may be used to access an image. Buffers are allocated in a round robin fashion, and need not be the same size, dimension, or datatype. This feature is especially useful for convolutions, block averaging, and similar operators.

(6) Fully automatic type conversion on both input and output. Conversion occurs only when data is accessed, so one need not type convert the entire image to access a subraster.

(7) IMIO implements general image sections (described below), coordinate flip, and subsampling.

(8) The dimensionality of the image expected by the applications code and the actual dimension of an image need not agree. If an operator expects a one dimensional image, for example, it may be used to operate on any line, column, or pillar of a three dimensional image, on both input and output (see discussion on image sections below).

(9) Both "compressed" and "block aligned" storage modes are supported, with IMIO automatically selecting the optimal choice during image creation (if the packing efficiency is not above a certain threshold then image lines are not block aligned). The device blocksize is determined at runtime and devices with different blocksizes may coexist.

(10) IMIO may be advised if i/o is to be either highly sequential or highly random; the buffering strategy will be modified to increase i/o efficiency.

(11) Pixel storage files may reside on special devices if desired. For example, the current **display** routine accesses the image display device as a random access imagefile via the standard IMIO interface. This was easy to do because FIO is device independent and allows new devices to be interfaced dynamically at run time (other examples of special "devices" are the CL, magtapes, and strings).

(12) The image header file, which is small, is normally placed in the user's own directory system. The pixel storage file, on the other hand, is often very large and is normally placed in a different filesystem. This is transparent to the user, and has the advantage that bulk data does not have to be backed up on tape when the user disk is backed up, and throughput is often higher because the pixel filesystem can be optimized for large transfers and more nearly contiguous files.

(13) An image opened with the mode "new_copy" inherits the full image header of an existing image, including all user defined fields, minus the pixels and minus all fields which depend on the actual values of the pixels.

The basic i/o facilities are described in the crib sheet. In short, we have procedures to get or put pixels, lines, or sections. The put calls are identical to the get calls and all buffer allocation and manipulation is performed by IMIO. The pixel access routines access a list of pixels (described by one, two, or more integer arrays giving the coordinates of the pixels, which are fetched in storage order to minimize seeks). An additional set of calls are available for accessing all of the lines in an image sequentially in storage order, regardless of the dimensionality of the image (as in the FITS reader).

4. Planned Enhancements to IMIO

The following enhancements are currently planned for IMIO; they are arranged more or less with the highest priority items first. The DBIO header, boundary extension facilities, and bad pixel list features are of the highest priority and will be implemented within the next few months.

(1) Replacement of the current rather rigid binary header by the highly extensible yet efficient DBIO header.

(2) Automatic boundary extension by any of the following techniques: nearest neighbor, reflection, projection, wrap around, indefinite, constant, apodize. Useful for convolutions and subraster extraction near the boundary of an image.

(3) Bad pixel list manipulation. A list of bad pixels will optionally be associated with each image. The actual value of each "bad" pixel in the image will be a reasonable, artificially generated value. Programs which do not need to know about bad pixels, such as simple pointwise image operators, will see only reasonable values. IMIO will provide routines to merge (etc.) bad pixel lists in simple pointwise image operations. Operators which need to be able to deal with bad pixels, such as surface fitting routines, will advise IMIO to replace the bad pixels with the value INDEF upon input.

(4) Implement the pixel access routines (**imgp__** and **impp__**). Currently only the line and section routines are implemented. The section routines may be used to access individual pixels, but this involves quite a bit of overhead and disk seeks are not optimized.

(5) Optimization to the get/put line procedures to work directly out of the FIO buffers when possible for increased efficiency.

(6) IMIO (and FIO) dynamically allocate all buffers. Eventually we will add an "advice" option permitting buffers to be allocated in a region of memory which is *shared* with a bit-mapped array processor. The VOPS primitives, already used extensively for vector operations, will be interfaced to the AP and applications sofware will then make use of the AP without modification and without introducing any device dependence. Note that CSPI is currently marketing a 7 Mflop bit-mapped AP for the VAX, and Masscomp provides a similar device for their 680000 based supermicro.

(6) Support for the unsigned byte disk datatype.

Long range improvements include language support for image sections in the successor to the SPP (subset) language compiler, and extensions for block storage mode of images on disk. Currently all images are stored on disk in line storage mode (i.e., like a Fortran array).

5. Image Sections

Image sections are used to specify the region of an image to be operated upon. The essential idea is that when the user passes the name of an image to a task, a special notation is employed which specifies the section of the image to be operated upon. The image section is decoded by IMIO at "immap" time and is completely transparent to the applications code (when a section is used, the image appears smaller to the applications program). If no section is specified then the entire image is accessed.

For example, suppose we want to display the image "pix" in frame 1 of the image display, using all the default parameters:

```
cl> display pix, 1
```

This works fine as long as "pix" is a one or two dimensional image. If it is a three dimensional image, we will see only the first band. To display some other band, we must specify a two-dimensional *section* of the three dimensional image:

```
cl> display pix[*,*,5], 1
cl> display pix[5], 1
```

Either command above would display band 5 of the three dimensional image (higher dimensional images are analogous). To display a dimensional image with the columns flipped:

```
cl> display pix[*,–*], 1
```

This command flips the y-axis. To display a subraster:

```
cl> display pix[30:40,310:300], 1
```

would display the indicated eleven pixel square subraster. To display a 2048 square image on a 512 square display by means of subsampling:

```
cl> display pix[*:4,*:4], 1
```

6. Use of Virtual Memory

The current implementation of IMIO does not make use of any virtual memory facilities. We have had little incentive to do so because 4.1BSD Berkeley UNIX does not have a very good implementation of virtual memory (few systems do, it seems - DG/AOS, which is what CTIO runs, does not have a good implemenation either). Various strategies can, however, be employed to take advantage of virtual memory on a machine which provides good virtual memory facilities.

One technique is to use IMIO to "extract a subraster" which is in fact the entire image. The current implementation of IMIO would copy rather than map the image, but *if* no type conversion were required, if no section was specified, if the image was not block-aligned, and if referencing out of bounds was not required, IMIO could instead map the image directly into virtual memory. This would be an easy enhancement to make to IMIO because all data is accessed with pointers. The code fragment in the following example demonstrates how this is done in the current version of IMIO.

```
int     ncols, nlines
pointer header, raster
pointer immap(), imgs2r()

begin
        # Open or "map" the image.  "Imagefile" is a file name
        # or a file name with section subscript appended.

        header = immap (imagefile, READ_ONLY, 0)

        ncols  = IM_LEN (header, 1)
        nlines = IM_LEN (header, 2)

        # Read or map entire image into memory.  Pixels are
        # converted to type real if necessary.

        raster = imgs2r (header, 1, ncols, 1, nlines)

        # Call SPP or Fortran subroutine to process type real
        # image.  Note how the pointer "raster" is dereferenced.

        call subroutine (Memr[raster], ncols, nlines)
            ...
```

Another, slightly different approach would be to allocate a single FIO buffer and map it onto the entire file. This would require no modifications to IMIO, rather one would modify the "file fault" code in FIO. This scheme would more efficiently support random access (to image lines or subrasters) on a virtual machine without introducing a real dependence on virtual memory.