IRAF Development Roadmap: 2014-15

Final Draft: 9/25/13

NOAO will begin an era of new operations at the beginning of FY16 as a result of widely publicized changes following the NSF/AST Portfolio Review. The NOAO Science Data Management (SDM) group will evolve substantially with the rest of NOAO to meet new challenges and opportunities in this new era. As a result, NOAO plans to address several aspects of the IRAF development environment prior to FY16 in order to place it on a sound footing during the period when SDM begins to focus on new initiatives.

IRAF development during 2014-2015 will focus generally on enhancements that will improve the scripting capabilities of IRAF and its performance on modern hardware. The goal is to have a major development push before FY16 that will be able to leave the system ready for new science application development and require only minimal platform support. We plan a number of related small projects that will make the system more efficient for in-house use as well as make the science codes more easily available to developers and applications outside the standard CL environment, specifically the Python programming language.

The development projects we propose break down into the following areas:

CL Language Enhancements	Improvements to the CL scripting language meant to create more modular (and thus more maintainable) tasks. ***[Sec. 1]***
In-Memory CL Image Operators	For rapid prototyping of image operations and to make more efficient use of modern hardware capabilities. ***[Sec. 2]***
Spectral Package Enhancements	Add the ability to handle spectra stored in the increasingly common table format. ***[Sec 3]***
Parallel Execution	To exploit multi-core/CPU hardware when doing data-parallel processing. ***[Sec 4]***
Python Task Interface	Re-use science tasks in Python environment or to develop IRAF tasks in Python. ***[Sec 5]***

Not all of these projects are the same size in terms of work required, indeed a project such as *'image operators'* implicitly requires additional work be done to have a new FITS image kernel as well in order to succeed. These requirements, implicit and explicit, are detailed below along with example of new functionality each project will offer.

1. CL Language Enhancements

The CL scripting language was originally envisioned as a means to write small programs that automate common tasks in the operating environment; indeed in the core

system the average script task is less than 100 lines long. However, users quickly adopted it as a means to write complete reduction pipelines thousands of lines long. This created a false reputation for the CL as being an unmaintainable programming language. Although it was never intended to be a complete programming language, the CL has become the way most higher-level tasks are now written (e.g. in the Mosaic/NEWFIRM packages and NOAO pipelines).

Lengthy scripts are often a byproduct of duplicate code, a desire not to create new tasks in a package and poor mechanisms for passing data between tasks. We plan to add sub-procedures to the scripting language as a means to allow lengthy tasks to be written in a more modular, and thus more maintainable, form.

1.1 Script-Task Subroutines

Script tasks currently allow one procedure statement per file, where there is a one-to-one mapping between the file defining the code for the task and the procedure which implements the task. By adding subroutines to this file, we allow the CL to explicitly declare a task as being the first procedure found in the script file, but create an execution context when running the script that recognizes other procedures in the script as being implicitly-declared tasks available *only* during the execution of the parent task (i.e. similar to *static* functions in most languages). For example,

```
procedure hello (who)
string    who           { prompt="use name"         }
begin
    print ("Hello '%s'\n", who)
    goodbye (who)
end

procedure goodbye (arg)
string    arg           { "", prompt="Adios who?"   }
begin
    printf ("Goodbye from subroutine, %s\n", arg);
end
```

One would declare a task HELLO using the command

```
cl> task hello = home$hello.cl
```

When the task is executed, the parser will find the *goodbye()* procedure and arguments and create a fake internal task declaration and parameter file. The *hello()* task then executes and is able to access the GOODBYE task as if it had been declared as a separate task within the package. Once the script terminates, the subroutine tasks are removed from the execution environment and are no longer visible to the rest of system. Only the toplevel task procedure is visible to any other task, e.g. a task FOO can call HELLO, but not GOODBYE in the above example.

1.2 User-Defined Functions

While subroutines add modularity to the language, they don't offer much in terms of passing back data to a calling procedure for re-use other than through the parameter

mechanism. By adding a type prefix to the procedure declaration we can create user-defined functions that return a single value and make the calling process more compact. For example,

```
procedure foo (val)
int    val              { 0, prompt="input value"   }
begin
   int  mysquare()

   printf ("Square of %d is %d\n", val, mysquare(val))
end

int procedure mysquare (arg)
int    arg
begin
   return (arg * arg)
end
```

Note that even if the *mysquare()* function were declared as a task itself it wouldn't currently be able to return a value directly, for instance the code in this trivial example would look like:

```
procedure foo (val)
int    val              {0, prompt="input value"    }
begin
   mysquare (val)
   printf ("Square of %d is %d\n", val, mysquare.result)
end

procedure mysquare (arg)
int    arg              { prompt="input argument"   }
int    result           { prompt="result value"     }
begin
   result = (arg * arg)
end
```

As with many procedural languages, the subroutine is appropriate to return multiple values (through the task parameters), while a function is more appropriate for a single value. Since any datatype supported by the CL can be made a function one can imagine functions being used to manipulate processing lists, strings (e.g. filename extensions), and so on.

1.3 Implementation Notes

Implementing both subroutines and user-functions are simply a matter of extending the CL grammar to allow the new syntax. The CL already has a concept of "*fake*" tasks (e.g. the root CL package), so declaring subroutines is a straightforward re-use of code. For user-functions we will need to make the array of built-in functions (the only functions currently allowed) extensible, and a syntax modification to the *task* statement will be required to declare functions stored as a single-procedure .cl file.

1.4 Work Justification

This project will allow scripts to be written in a more modular form, reducing the number of bugs and leading to greater code re-use. It also creates a syntax that is more natural for programmer and may lead to a renewed adoption of CL scripting for task development outside of NOAO. We see its primary benefit as simplifying development of tasks in the NOAO pipeline systems.

2. In-Memory CL Image Operators

The task-based nature of the CL makes for some unnecessary coding structure, e.g. to create image lists or loops over extensions in an MEF file, the creation of intermediate files, and so on. Tasks like IMEXPR can help to some extent but don't handle multi-extension data well in all cases. Logically one wants to write expressions or perform operations on an image without having to worry about the nature of a task. We plan therefore to add a new datatype to the CL language called an *image operator* to allow this. For example,

```
cl> image  a, b = "object.fits", c="flat.fits"          (1)
cl> a = b / c.median(5,5)                               (2)
cl> string bias = a.keyword ("BIAS")
       ...lots of other work
cl> a.write ("result.fits")                             (3)
```

On line (1) we declare three image operators `a, b` and `c` where only the first operator is uninitialized. On line (2) we write an expression that sets the result to the '`a`' operator as the value of the '`b`' operator divided by the 5x5 median of the '`c`' operator. The *median()* operation borrows from the Object Oriented syntax concepts and in the evaluation of the expression would create an intermediate result used as the divisor (other operator methods, e.g. *a.len()* where '`a`' is a list, will also be implemented where it makes sense). The last line shows the *write()* method for dumping the result to a disk file. A complete list of operator methods would look similar to the list of builtin functions already permitted by the IMFUNC/IMEXPR tasks.

Compare this to how such an operation would be done today:

```
cl> median ("flat.fits", "temp.fits", 5, 5)
cl> imarith ("object.fits", "/", "temp.fits", "result.fits")
cl> imdel ("temp.fits")
```

and so on, with the situation being worse if the data were in MEF format and we had to loop over the extensions explicitly. Note we could not simply use IMEXPR in this case because of the need for a 5x5 median: IMEXPR has a *med()* function but it only works in one dimension.

A slightly modified version of this example would use the *image pointer* operator as follows:

```
cl> image  a, *b = "object.fits", *c="flat.fits"        (4)
cl> a = b / c.median(5,5)                               (5)
```

```
cl> a.write ("result.fits")                                    (6)
```

The difference here is that the 'b' and 'c' operators, by virtue of being pointers, will load the entire image in memory *at the time the operator is declared*. In our previous example the evaluation of 'a' will read the named 'b' and 'c' files *at the time that statement is executed* (and create a temp file for 'a'), perhaps also loading the data in memory and then releasing the resource when done. If another statement modifies the file e.g. '*flat.fits*' then in the first example the expression always evaluates with the latest contents of the file, however when declared in-memory with a pointer the disk file may differ from the pixels in the operator. An operator *sync()* method can be used to resolve this difference, but it is also possible to exploit this behavior in a task.

The primary use for an image pointer operator is to force frequently used images to stay resident in memory and thus reduce processing time. To some extent the system memory cache will also do this, but with no direct control by the user over when/whether the data are swapped to disk. Because the CL is built on IRAF interfaces the same as any other task, in-memory operators should also work on older IMH format images (with proper data structures that identify the kernel to be used). It is worth noting that tasks like IMEXPR/IMARITH are used in hundreds of script tasks (some 80% of those cases are in external packages developed outside NOAO), while we do not expect these to be re-written to use image operators, it does show that image expression evaluation is a common task.

The discussion of image operators to this point has focused on their use within the CL, but consider the case where we want to call a compiled task. At present we can do something like:

```
string imname = "foo.fits"
imarith (imname)
```

Logically then we would also wish to be able to write:

```
image img = "foo.fits"
imarith (img)
```

There are two ways this can be implemented given that the expected parameter type is a string - they both involve some invisible magic. Assume our image operators have methods to access details of the operator, specifically let's call them *fname()* and *addr()* to get the underlying file name and the in-memory address (as a string) respectively. Knowing this it is possible to do something like

```
imarith (img.fname())   or    imarith (img.addr())
```

This is clumsy for the astronomer to write in their script but would be an acceptable first solution. However, as we parse the statement we can compile the intermediate code as though it were written using the clumsy methods and achieve the more elegant solution for the script writer transparently.

If we compile the `img.fname()` method we can operate on the disk file as always, using the `img.addr()` method will require the ability to map memory addresses as images. The choice of which method to use will depend on whether the operator is declared as "`image img`" or "`image *img`" per the discussion above.

2.1 Multi-Extension FITS, Image Sections and Operators

Script variables, specifically strings, are commonly used to hold image names. When extension numbers or image sections are required these are concatenated to the string using something like

```
cl> string str, img = "foo.fits"
cl> int    extnum = 1
cl> str = img // "[" // extnum // "]"
```

to create the string "foo.fits[1]". Other concatenations to create *@-files*, add image sections, loop over extension numbers, get a named extension are also commonly used.

The same invisible magic trick above to use the *fname()* and *addr()* methods can be implemented here to allow the same use of concatenation for image operators. When the address method is used, the underlying image kernel code will need to be able to apply the extension or section specification to the start address, this change shouldn't be dramatic in the code (certainly no more than was required to handle URLs in the system).

2.2 New FITS Kernel

For in-memory operators to work, we'll need a new version of the FITS Kernel based on CFITSIO. Not only does this get us access to compressed FITS images (and other conventions), but we are also able to use the feature of CFITSIO that allows us to map in-memory data. Although apparently only mentioned in passing here, it is important to note that implementation of a new FITS kernel will likely require the majority of the time needed to implement this feature.

More specifically, a FITS image would be mapped in memory in such a way as to separate the PHU, EHU and data records to avoid the complications of dealing with a single file and I/O contention issues. This would mean that each extension could be manipulated separately while in memory, independent of file synchronization concerns (important below). We plan to use the Shared Memory Cache (SMC) interface developed for the Mosiac/NEWFIRM/[CK]OSMOS DHS systems to handle in-memory data.

2.3 Work Justification

This project will allow script developers to directly deal with images in a natural way, additionally giving them control over which images reside in memory as a means to tune performance. When used with the parallel execution project (see below), the proposed scheme will allow extensions within a large MEF file to be used independently. We expect the primary benefit will be to CL users (writing scripts or working interactively) doing rapid prototyping of analysis routines and to improve the performance of the NOAO pipeline tasks.

3. Spectral Package Enhancements

The NOAO *onedspec* format relies largely on image header keywords to define the dispersion with the pixel values representing the intensity at each wavelength. An alternative format used more frequently is to store the data in a table with columns representing the wavelength, intensity and optionally an error value; this table can be

stored in a variety of file formats include FITS BINTABLE extensions, native XML or VOTable. The majority of the work needed to support tabular spectra was completed a while ago but never completed and integrated into the system, this project is intended to finish that initial work and add conversion tools and utilities necessary to support multiple formats and instruments.

The first step is already underway: the work on the tabular support done in 2001 and subsequent updates to the ONEDSPEC and APEXTRACT packages are being merged into what will become the new version of the tasks. This will be assembled into a new XONEDSPEC external package that will serve as the development and testing code base while the final work to support tables is completed. This will allow us to distribute a significantly new version of the code that can have an update/release schedule independent of the core system. ESO is planning to use this new capability in the backend to their archive system and will exercise the code thoroughly. Additionally we plan to work with them to define the needed format conversion tasks as well as add new tasks to the VO package for dealing with spectra from the Virtual Observatory.

3.1 Work Justification

This project is needed primarily to allow IRAF to work with spectra from non-NOAO instruments that use tabular storage formats, both to increase the utility of IRAF in the general user community and for in-house use as part of the NOAO/LSST Data Lab. This fills a gap in software able to work with spectra available in the Virtual Observatory, and provides a framework for supporting a wider variety of data. We expect the primary benefit to be to software developed in support of the DESI project (by individual scientists or science collaborations) and for use within the NOAO/LSST Data Lab

4. Parallel Execution

Modern desktop/laptop systems now routinely come with multi-core or multi-cpu capabilities, often times with GPU processors as well. Likewise, instruments such as DECam and ODI are producing MEF files with many extensions, creating an embarrassingly parallel data-reduction problem. IRAF tasks have the ability to process lists of images easily, and lists of extensions just as easily, however this is always done in a serial manner regardless of the hardware available. For non-interactive tasks, advanced users have invented scripts that create multiple background jobs to process lists in parallel that make the most of their hardware. This background mechanism is the only level of parallelism currently available for execution (ignoring the multi-processes involved in the CL-task interaction, graphics, image displays, etc).

The NOAO pipeline system farms out files or extensions to multiple nodes to achieve the desired parallelism and improve performance, however this is a large framework for single-user systems. To achieve the same results, we would like the CL to optimize task execution where possible, or at a lower level, to allow SPP compiled tasks to use a threading interface to parallelize themselves. The chief obstacle to a threading interface is the implementation of the I/O interface in the VOS that would require substantial changes (and thus a possibility of new bugs) to overcome. Some experimentation with GPU hardware might also be considered if a specific need is identified for optimization.

4.1 Parallel CL Evaluation

Consider the image operator example above where the image operators refer to MEF files with many extensions (e.g. an ODI frame):

```
image *a, *b="object.fits", *c="flat.fits"
a = b / c
```

Because the operators are *in memory* (using the SMC structure described above) and this is strictly an evaluation of pixels, the 'b' and 'c' operators are read-only and the 'a' operate is created on the fly. In this case we *can* use threads to parallelize the evaluation since we bypass the I/O limitations by working on unique memory data addresses that will be resolved to a single file-store when the script terminates anyway.

There are, of course, complications depending on the expression involved, e.g. what if 'c' is a constant float value or a single-CCD FITS file to be divided into all extensions of 'b'? This will be resolved in the evaluation code automatically to evaluate only *correct* expressions (i.e. those expressions involving constants, images of the same dimensionality, etc) or else throw an error (this same checking is done in the IMEXPR/IMARITH tasks and so simply migrates to the CL code evaluation).

The evaluation itself will use the same intermediate-code rewriting trick as above, either compiling to a loop structure or more simply a *parallel_eval(<expr>)* or some such internal function to do the work.

4.2 Parallel Task Execution

Parallelizing task execution is more difficult at a systematic level since it is impossible to infer programmatically from the parameters what the correct action should be. There are however two things that might be done if we determine there are significant speed gains that could be achieved:

In the first case it might be possible to introduce a new task (or several) to do execute a command in parallel using the current backgrounding mechanism. This would allow the programmer to have more control over how the parallelization is done by specifying which parameters can correctly be split for execution. Thus, a command such as

```
imarith (b, "/", c, result=a)
```

(where 'a', 'b' and 'c' are MEF files or @-files of image names) would execute as something like (in *pseudocode*):

```
for (i=0; i < len; i=i+njobs) {
    for (j=0; j < njobs; j=i+) {
      imarith (b[i+j], "/", c[i+j], result=a[i+j) &
    }
    wait                                  # wait for completion
    njobs = min (njobs, (len-i))          # update job counter
}
```

Note that this approach has the added benefit of being able to apply lists (or MEFs) to tasks that don't already accept list input.

A second approach is to implement a threading interface in the IRAF kernel for purely computational code, then retrofitting specific tasks that would benefit the most. It is unclear whether this would have any clear benefit to tasks already in the system, however for the purpose of developing new compiled tasks such a threading interface could have a major benefit.

4.3 Work Justification

This project is needed to optimize the performance of IRAF on modern hardware when dealing with large-format mosaic images (e.g. Mosaic, NEWFIRM, ODI or DECam) or when processing a large list of images (e.g. in a nightly pipeline). We expect the primary benefit to be to the average desktop user doing general image processing, the NOAO pipelines, and for use in supporting DES/DESI and the NOAO/LSST Data Lab.

5. Python Task Interface

Even with the significant changes to the CL programming language proposed here, Python will continue to gain interest in the community as a platform for new software development. Projects such as AstroPy are working on creating a community-built package of libraries but are nowhere near ready to offer even basic applications that will replace IRAF.

Our goal with a Python interface should serve two purposes:

- Allow pure Python applications a clean and "pythonic" method to call IRAF tasks,
- Allow pure Python applications to be called from within the IRAF environment using the CL task syntax.

The first goal allows programmers to make use of the science code available in IRAF from their chosen environment in cases where there isn't a native python solution. The second goal allows for development (or use) of IRAF tasks in "non-IRAF" languages, making it easier to create IRAF packages with more diversity. In both cases we strive for *interopability* between tasks more than *integration* of one environment into the other. Additionally, we can use real-world examples of PyRAF tasks to get a sense of what features are important to developers.

5.1 Implementation Strategy

Our strategy is to rely on the concept of a task as the basis for interaction, e.g. a python script wants to call an IRAF task to perform some action on an image, or the CL wants to execute a task implemented as a python script, in much the same way either system would invoke a host command as a task. A task has parameters passed in as input, has the ability to pass back results (the text output, a FITS file, etc) and operates within some defined environment.

To do this we plan to use JSON-RPC as a means to drive the CL to execute a single task from Python as a remote procedure, the task I/O and graphics streams can be redirected as needed depending on the usage. Similarly, from within IRAF we would execute a Python script as we would any other foreign command (e.g. unix host commands, IMFORT tasks) with the exception that we'd need to establish the proper environment in

python first. In both cases we require only that each environment be available for the task to run, i.e. a python script would require that IRAF be installed on the machine (or vice versa) but we make no attempt to bundle an IRAF and Python system into a single distribution, we leave that up to the user or a development project building a distribution.

This use of a Remote Procedure Call (RPC) interface has several advantages:

- There is a clear separation between python concerns of a script and how it might be implemented in IRAF.
- A JSON-RPC command interface in the CL means that we are not tied to python as our only language option, interfaces to 'R' or Javascript could just as easily be developed in the future.
- A tasking interface is more easily able to create a native python (so-called 'pythonic') interface, e.g. one which accepts python lists or dictionaries as input rather than one implementing IRAF idioms such as file lists and parameter sets.
- Any python interface we develop can either be adopted and extended by the community, or replaced entirely by something more acceptable to the python purists, without requiring further work on the IRAF side.
- IRAF runs entirely in its own process space, again making the purists happy.

The CL-side running the JSON-RPC server is straightforward to implement and similar to what's already been done for the SAMP interface, some investigation will be needed to see whether the issue of redirecting the I/O and graphics streams, cursors, etc are an issue for real-world applications.

The python interface itself can also be designed independently. It will need to manage/sync the environment (e.g. sending the current working directory, environment variables, etc), sending the task parameters and controlling the execution however the steps required are already well understood. Other than requiring a specific version of IRAF that supports the RPC interface we plan to develop this interface as an independent project, perhaps even as an affiliated AstroPy package (if they'll have us).

It is beyond the scope of this document to detail this interface here, however we believe we understand the issues well enough to proceed. There are, of course, limitations with this approach (e.g. passing back cursor reads from the IRAF task to the python script), however even these could be overcome with sufficient effort if required. Our examination of PyRAF uses does not indicate these limitations would prevent any of these PyRAF tasks from working in the same way using this interface. This same architecture is being used to develop a tasking interface for the VOClient tools implemented under the VAO grant, we expect much of that work will inform and support the work here.

5.2 Python Task Development

For python scripts we would like the call to an IRAF task to be as easy as it is to connect to a web interface now, e.g.

```
import iraf

iraf.init ()
```

```
try:
  data = iraf.cl.osfn ("data$")
     res = iraf.imstat ( ["dev$pix", data+"/*.fits"] )
except TaskException, e:
     print e.error
iraf.shutdown ()
```

In this example the *init()* method would either start a CL session running or else connect to an existing one and synchronize the needed environment (we don't show how one might capture an error if no CL is available). It would get the list of loaded packages and available tasks and dynamically bind the IMSTAT task. We can resolve IRAF logical paths (i.e. the 'data' path) and when the *imstat()* method executes the interface would take the python list argument and convert to the needed types in the task parameter set and start the task running. In this example we capture the *stdout* stream as a result string, however we could also allow the output to be printed directly to the screen. The final *shutdown()* method would close the CL connection and shut down the server.

The syntax in many ways will be similar to what is already seen in PyRAF, making conversion of existing tasks trivial. Note however that we now allow python objects such as the image list to be used when calling the task, something PyRAF does not currently allow. The connection to the CL server and the task execution are all hidden from the programmer and contained in the *'iraf'* interface itself yet are accessible if needed. For the programmer we are able to document the usage and exception handling to present a clear interface for both high-level simplified usages for casual developers, and the low-level interface needed for finer control of the system.

5.3 IRAF Task Development

IRAF tasks already have the concept of a 'foreign' task, i.e. one that executes in a shell created for the command (a common example is the use of the *'ls'* Unix command as transparently as the IRAF 'dir' equivalent). In the case of python scripts we would simply invoke the python interpreter instead of a Unix shell and automatically convert the task parameters to command-line arguments. For example, consider the trivial python task stored in the file *'hello.py'*

```
#!/usr/bin/env python

import sys

print "Hello", str(sys.argv[1:])
```

This takes a single argument and could be executed as

```
% hello.py world      or    % python hello.py world
```

In an IRAF package this script and an associated parameter file would live in the same directory. The package CL script would define the task as a python script in some manner so that when it was executed the parameter file would be found and the python script executed with the proper arguments. If the script actually made calls to IRAF tasks as

above, it would simply reconnect to the parent CL to initialize the same as if it had been started from outside the CL.

5.4 Comments

We believe our approach justifies new development rather than simply adopting PyRAF for a number of reasons:

- Our approach provides a clean separation between python and IRAF through remote procedure interface, avoiding complications of mixing the environments and process space,
- We address the python purist community's concerns about PyRAF,
- We don't plan to require any third-party code dependencies,
- We allow for interfaces to languages other than python in the future,
- We provide a means for python developers to extend the interface on their own and as they see fit.

5.5 Work Justification

This project is needed to protect the large investment already made in IRAF so that time-tested tasks can continue to be used in modern programming languages, and so that modern languages can be used to create new tasks. We expect the primary benefit will be to an IRAF system that can attract new interest from a user community not put off by what they see as '*old*' technology, and more generally to the community that doesn't have to reinvent science applications that they have come to trust.