MSRI HOPS Re-development Specification

Geoff Crew and John Barrett
MIT Haystack Observatory

Rough Draft Version 0.1, May 7, 2020

Contents

1 Architecture

The goal of the re-developed HOPS architecture is provide a component-based design which allows for a greater degree of flexibility in the types of operations that may be applied to VLBI data during fringe fitting, and avoid the semi-monolithic approach used in the past. To do this, the software package to be delivered will primarily comprise of a set of loosely coupled libraries which can be composed into an variety of calibration and fringe-fitting applications. Moreover, rather than attempting to identify and categorize all manner of data and data manipulation which may be required by future observations at the outset, a primary goal of this project is to provide a relatively flexible set of data types and a plugin interface so as to allow for future extensions with minimal revision to the existing code.

For the foreseeable future, the main source of input data to this software will be provided by the DiFX correlator. However, this may not always be the case, so in order to decouple the correlator from the post-processing software a conversion utility must be provided. This follows the same paradigm as the current code (difx2mark4), except that we propose that this conversion utility operate mainly as a transparent pass-through, merely converting the correlator output into the native post-processing format, rather than applying any initial calibration (e.g. auto-corrs) or corrections.

Once the data has been converted to the native post-processing format additional manipulation will take place in several stages, such as data-flagging, a priori phase/delay calibration, fringe-fitting, and post-fringe-fitting calibration. Finally the data will be made available for export to archival formats (e.g. HDF5).

TODO - expand on this, add diagram of data flow and add diagram of control/data-flow in 'fringe-fitter'.

2 Language, Build and Version Control System

Several aspects need to be taken into account when deciding on a choice of programming language for this project. Namely, some of these are:

1. Availability of software developer expertise.
2. The inherent performance attainable with a specific language.
3. Availability of high performance open source utility libraries for math, I/O, etc.
4. The primary language of the existing code base (C).
5. The accessibility and ease of extensibility of the project by users with varying levels of experience

Obtaining a reasonable balance between these considerations is difficult with a single language. Therefore we plan to develop a multi-language project, wherein the base computation layer is handled within C/C++, but additional data manipulation can be done via optional Python plugins embedded within the aapplication or independently by external Python scripts which have access to some of the underlying application libraries. C++ is a good choice given current personnel and developer expertise, and being a super-set of the C language it provides the ability to reuse of portions of the existing C code base with little to no change, while also adding modern language features (templates, classes and inheritance, const. correctness, function overloading, etc.). A combined C/C++ approach allows for much easier memory management (currently handled rather painfully in the existing HOPS code base) and also enables the use of a wide variety of open source libraries, not least of which is the built in standard template library which provides access to a wide collection of basic data types (strings, vectors, maps, etc) and algorithms (searching, sorting, etc.) which will reduce the required amount of maintenance of internal code and reliance on external libraries.

Further augmenting C/C++ libraries with inter-language communication to Python can be done via a wide variety of mature tools (ctypes, boost.Python, SWIG, pybind11, etc.), and will increases the ease at which outside users can augment the software. Since Python 2 is no longer supported, all new development will be Python 3.

The build system for this project will be the CMake build system [1], and version control will proceed through a locally hosted (Haystack) git repository. The CMake build system is generally easier to maintain than the current autotools system used by HOPS when faced with the complexities of a multi-language project. It also

[1] `https://cmake.org`

allows for a more user-friendly configuration at the time of compilation, as the user can be presented with a menu providing options which are dependent on the available set of tools/libraries that are currently installed and detected on the users system. It is also suggested to move to the git version control (over Subversion) system for several reasons[2]. Two of which that particularly stand out are:

1. Its access control model lets the repository owner maintain local control over the code base while providing and the ability to accept changes from external entities without them needing permissions. This will make it much easier to leverage community contributions if so desired at some point in the future.

2. The manner in which branching and merging is handled in git is less cumbersome as it tracks the complete source tree. This is especially important when a project contains multiple independent tools that have separate development schedules. For example this obviates the need for a 'virtualtrunk' – the model used by DiFX.

3 Parallel processing

The existing fringe-fitting process is largely a data-parallel process operating on individual baselines with no inter-process communication. This lends itself easily to simple parallelism using multiple independent processes (SPMD), which has been exploited [**blackburn2019eht**] to deal with the EHT data volume. However, this approach eliminates the ability to simultaneously fit for global or station-based parameters and requires multiple iterations in order to apply successive calibration/corrections. Therefore, if some calibration tasks are to be done simultaneously with fringe-fringe fitting, this will require both a substantial architectural change from the current fitting algorithm, but also necessarily reduce the degree of (simple) parallelism available. To accommodate this, some parallel processing will have to be addressed within the application. To do this we will use a multi-staged approach during the course of development.

Initially, the software to be developed under this project will focus on a simple single-threaded implementation, whereupon careful profiling will inform us of the largest computational bottlenecks. For example, in the current HOPS code the majority of the computation time is spent in a single routine (vrot.c, which is essentially just multiplying two complex numbers) that is applied over a large array of data. This sort of task can easily be parallelized on modern multi-core architectures using the SIMD approach. We propose to use OpenCL for this purpose given its support on a wide variety of architectures (mult-core CPUs, GPUS, hardware accelerators, etc.).

This type of parallelism exploits vector instructions and/or multiple processors to execute the similar operations over wide swaths of data. while given current experience with the existing HOPS leads us to expect that SIMD parallelism should largely be adequate to accelerate fringe fitting, if during the process of development it is discovered that a thread-based model (MIMD, where each parallel thread follows a semi-independent code pathway) could be more advantageous, then we propose to use OpenMPI framework to implement this. The reason being, that while OpenMPI does have a larger development overhead than most other threading libraries (e.g. pthreads, c++11 threads, etc.) it can be easily scaled beyond a single computer when and if needed. However, it should be noted that by necessity, any OpenMPI implementation must be developed as an independent executable (although it would be able to take advantage of any available libraries and SIMD policies present).

4 External Package Selection

To the extent possible the software to be developed in this project should make all external dependencies optional whenever possible. That is to say, that while some features may be missing if an optional external package is missing, the core functionality of the software should not be affected. For example, if the fast Fourier transform library FFTW3 is missing, the code should fall back to an internal implementation (which is allowed to be slower), but which produces equivalent output. Likewise, while an effort should be made to support portability of the output data into useful downstream formats (HDF5), this should not preclude the use of a native format capable of storing the same data for development and debugging purposes.

[2] `http://git.wiki.kernel.org/index.php/GitSvnComparsion`

The following table is a preliminary list of which external packages are expected to be incorporated as required or optional dependencies. If an optional package is missing on a user's machine the software will default to a fallback option (if available) or disable the features which require that particular package.

Name	Optional/Required?	Purpose	Fallback option
C++11 STL	Required	Standard template library	None
Python	Required	Scripting language interpreter	None
pybind11 or boost.Python	Required	C/C++ ↔ Python interface and bindings	None
OpenCL	Optional	SIMD parallelism	Single-threaded implementation
OpenMPI	Optional	MIMD parallelism	Single-threadd implementation
FFTW3	Optional	fast Fourier transform acceleration	TBD native code
GSL	Optional	Linear algebra and special functions	TBD native code
HDF5	Optional	File input/output	TBD native binary format
PLPlot	Optional	plotting library	None
difxio	Required	DiFX I/O library - file converter only	None

Table 1: List of optional/required dependencies.

5 Objects specification

Generally speaking the code will be organized around roughly two object type categories involved in the structure of the new HOPS. These are as follows:

1. Data Containers: These serve to organize the visibility data and metadata associated with an observation.
2. Data Operators: These evaluate a function or perform some transformation on a given data container. Their operation may by directed by set of externally defined conditions (i.e. defined outside the data containers upon which they operate).

5.1 Data Containers

The existing HOPS code base relies on a fixed number of `C` structs to organize and present the data related to an observation. The strict memory layout of these structures has the advantage of making them cross-machine compatible, which is necessary since these structures are also used as the core components of the Mark-4 I/O library. However, a notable disadvantage of this rigid design is the degree of difficulty encountered in making changes to the existing data structures, or adding new data types in order to accommodate additional information which was not originally envisioned at the time the library was written.

To make the data structures more flexible we intend to decouple the in-memory data layout from the file I/O, so they do not necessarily need to be byte-for-byte copies. Furthermore, to the extent possible, the in-memory data structures should be classes which provide access via a key-value pair mechanism so as to avoid exposing the private internal storage layout to the routines needing access to subsets of the data.

In a strictly typed language such as `C`, flexible data structures have a high degree of code overhead, not only in the management of dynamic memory allocation, but more severely in the conversion of data types and typecasting. To ameliorate this we propose to exploit `C++11`'s variadic template mechanism, which allows for the transformation of type-agnostic class lists into concrete class types or hierarchies at compile time. This makes it possible to store disparate types (known at compile time) within in the same object that are indexed and can be retrieved by the same type of key (e.g. a name string).

We propose the following basic set of class templates be used to construct most in-memory objects used for the manipulation of correlated observation data and its associated metadata:

1. ScalarContainer - encapsulates scalar-like data with associated units
2. VectorContainer - encapsulates vector-like data with associated units
3. TensorContainer - encapsulates rank-N tensor-like data with associated item and axis (vector) units

4. HeterogenousContainer - encapsulates any of the above types

These template classes are to serve as a simple wrapper around the management of the raw memory needed to store a data item and keep track of its associated unit(s), and (if applicable) its axis values and their units.

Listing 1 gives a brief sketch (without implementation) of the class templates for these data container objects, while listing 2 gives a simple example of what template declaration of an object storing visibility data from an observation over multiple baselines and polarization products might look like.

```
template< typename XValueType, typename XUnitType >
class ScalarContainer {
    public:
        ScalarContainer();
        virtual ~ScalarContainer();
        //...TBD impl...
    private:
        std::string fName;
        XUnitType fUnit;
        XValueType fData;
};

template< typename XValueType, typename XUnitType >
class VectorContainer {
    public:
        std::string fName;
        VectorContainer();
        virtual ~VectorContainer();
        //...TBD impl...
    private:
        std::string fName;
        XUnitType fUnit;
        std::vector< XValueType > fData;
};

template< typename XValueType, typename XUnitType, size_t RANK, typename... XAxisTypes >
class TensorContainer {
    public:
        TensorContainer();
        virtual ~TensorContainer();
        //...TBD impl...
    private:
        std::string fName;
        XUnitType fUnit; //units of the data
        std::vector< XValueType > fData; //row-indexed block of data
        std::tuple< XAxisTypes > fAxes; //tuple of length RANK of VectorContainers
};
```

Listing 1: Data object templates

```
//forward delc. unit types
class CorrelatedFluxUnit;
class MegaHertzUnit;
class SecondsUnit;
class Unitless;

typedef VectorContainer<double, SecondsUnit> TimeAxis;
typedef VectorContainer<double, MegaHertzUnit> FrequencyAxis;
typedef VectorContainer< std::string, Unitless> PolarizationAxis;
typedef VectorContainer< std::string, Unitless> BaselineAxis;

class Visibilties: public TensorContainer< std::complex<double>, CorrelatedFluxUnit, 4, TimeAxis,
    FrequencyAxis, PolarizationAxis, BaselineAxis > {
//...impl..
};
```

Listing 2: Visibility object type

5.2 Data operators

The data operator classes are meant to organize the mathematic manipulations which are to be performed on the data containers. For example, many of the operations performed in the existing HOPS code-base (such as the application of a priori phase calibration) are relatively trivial linear transformations applied to the visibility data. However, they are currently intertwined with a large amount of control logic which obscures the basic data pathway (e.g see postproc/fourfit/norm_fx.c) Most unary or binary operations that are to applied to visibility or other data residing in TensorContainers such as scaling, multiplication, transposition, summation, Fourier transformation, etc. will be made available as individual classes inheriting from the same interface. A uniform class interface will allow these data operators to be composed or modified to create more complicated composite operators or strung together to accomplish data pipelines of arbitrary complexity. An additional advantage of encapsulating individual operations is that any SIMD parallel-processing extension used to accelerate data processing can be made opaque to the user. Listing 3 gives a brief sketch of the class templates generalizing the data operators.

```
class DataOperator {
{
  public:
    DataOperator(){};
    virtual ~DataOperator(){};
    virtual void Initialize(){};
    virtual void ExecuteOperation() = 0;
};

template< typename XInputType, typename XOutputType >
class UnaryDataOperator: public DataOperator {
    public:
        UnaryDataOperator();
        virtual ~DataOperator();
        void SetInput(const XInputType& input );
        void SetOutput( XOutputType& output);
    //....impl...
}

template< typename XFirstInputType, typename XSecondInputType, typename XOutputType >
class BinaryDataOperator: public DataOperator {
    public:
        BinaryDataOperator();
        virtual ~DataOperator();
        void SetFirstInput(const XInputType& input1 );
        void SetSecondInput(const XInputType& input2 );
        void SetOutput( XOutputType& output);
    //....impl...
}
```

Listing 3: Data operator templates

6 Interactivity definition

Interactivity with the user will proceed primarily through a python scripting interface. This avoids the need to develop a separate scripting language, as the python interpreter can be embedded in the application. Access to library objects will be exposed via python bindings, so that a user may have direct access to the data containers, and can implement external routines which can be insert as a data operator. Real-time interactivity is not planned to be implemented.

7 Libraries and executables

7.1 Libraries

The software to be delivered will be presented largely as set of libraries with several executables which utilize them. These libraries will be organized by their primary function and are expected to consist of the following:

1. Core - Common template and abstract base classes, messaging and other utilities.
2. Math - Common mathematical functions and minimization routines.
3. Data containers and native I/O - Definitions of the in-memory data objects and a simple binary I/O interface.
4. Data operators - Definitions of data operations
5. Builders and process management - Provides classes which constructs data operators and the pipeline which organizes them from user input.
6. Plugins:
   (a) HDF5 I/O - Converts the in-memory data objects to/from an archival HDF5 file.
   (b) Plotting - Utilities to generate plots for data exploration (e.g. fringe-plots).
   (c) Python interface - Interpreter for user scripts and python bindings to C/C++ objects.
   (d) SIMD extensions (OpenCL) - Parallel implementation of some set of data operators

7.2 Executables

The executables to be delivered will largely be composed through re-use of the library code and at a minimum will consist of the following (though not necessarily named as such):

1. DiFX2Input - difx2mark4 equivalent, converts correlator (difx) data into native input format.
2. Station utilities - converts station calibration data (e.g. ANTAB) files into native input format.
3. FringeFitter - fourfit-equivalent, accepts a user script for configuration and direction and applies it to the visibility data.
4. DataInspect - data inspection tool, dumps data objects to a human readable format.
5. FringePlot - plotting tool, creates fringe-plots and all graphical data exploration.
6. HDF5Export - converts any native formats into an HDF5 format for access by external programs.