Appendix A

The user interface for off-line applications

A.1 The User interface commands

The user interface of all application programs by default presents a list of keywords to which the application is sensitive. This is presented as a list of variables of an embedded interactive shell. The user can set, reset, load, save, etc. the values of these keywords using the basic shell commands in an interactive shell. In general, keywords can be assigned a list of comma-separated values. The number of values that a keyword expects can be found via the shell-command `'?'`. The value of a keyword can be set by a command of type `<KeyWord>=[<Val0>[,<Val1>[...]]]` and can be reset by omitting the values in such commands (command `<KeyWord>=`). The shell-command `help` provides the basic help about the shell itself and prints following on the screen:

```
Commands in the interactive mode:

Use <Key>=<Val1,Val2,..> to set value(s) for a keyword
Use <Key>=<RETURN> to unset value(s) for a keyword

 inp         : To see the various keywords and their values
 go          : To run the application
 gob         : To run the application in background
 cd          : Change working directory
```

```
 help        : This help
 ?           : Information on the type of the keyword
 explain     : Detailed help, optionally of keywords/task [[Key][:Task]]
 save        : Save the values, optionally in a file
 load        : Load the values, optionally from a file
 edit        : Use an editor to (un)set the values
 quit        : Quit the application

Any other input will be passed to the system shell
```

These commands form the standard set of shell-commands available from all applications. However, the user interface libraries provide mechanisms for application programmers to add application specific shell-commands as well as alter the behavior of the basic shell-commands. Hence, there may be a few extra commands which may be application dependent and will change from application to application. However, the `help` command will automatically always show all the commands available for a given application.

Following is a short description of the shell commands:

- Command `inp` displays the current setting of the keywords.
- Command `go` signifies the end of the interactive session. After this commands is issued, the values of the keywords set by the users are used by the application internally and the execution of the main program proceeds.
- Command `gob` is similar to `go` except that it will run the application in the background and print the process ID (PID) number of the background process on the screen (also see section A.2).
- Command `cd` changes the current working directory within the interactive shell.
- Command `'?'` gives some basic information about the type and the number of values the keywords expects.
- Command `explain` gives as detailed a help about the application and the keywords as the author of the application has cared to write in the help file. These help files are located in the directory specified by the environment variable `GDOC` (see section A.2).

- Command save saves the current setting of the keywords in a file. By default, these values are saved in a file named ./<Application Name>.def. If a file name is supplied as an argument of this command, the values will be saved in the file. this command.

- Command load is the conjugate of save - by default it loads the settings for the keywords from the file ./<Application Name>.def. If this file is already present when the application is started, it is loaded automatically. Alternatively, it can load settings from a file provided as an argument to the load command.

- Command edit allows user to edit the keyword values in an editor of choice specified by the environment variable EDITOR (see section A.2).

- Command quit quits the interactive session without executing the application.

All inputs to the interactive shell, which are not any of the above mentioned commands or any of the application specific commands, are passed to the underlying Operating System (OS) user shell. Hence, most of the native OS commands shall still be available from this shell. (Users however must be aware that certain OS shell commands like “setenv” (for csh users) or “export” (for bash/sh users) will seem to work, but will not have the desired effect).

As mentioned earlier, the values of the keywords can be a list of comma-separated values. The expected number of values will be reported by the shell-command '?' along with the type of the values, enclosed in '[' and '] ' pairs. A blank set of square brackets (“[]”) will be printed for keywords which can accept any number of values. Some keywords may accept values of mixed types (e.g., string, floats, integers). For such keywords, the reported type will be “UNKNOWN”.

Comma is treated as a separator in a list of values for a keyword. To suppress its interpretation as a separator, it can to be “escaped” using the backslash ('\') before the comma. For example, if the value of a keyword key has to be set to a string ‘‘Funny, value with a comma’’, it can be done using

```
key=Funny\, value with a comma
```

The characters '\', '[', ']', '=', which are part of the shell-command syntax, must also be escaped similarly to be used as part of the values of the keywords.

Numeric values of keywords can be arbitrary mathematical expressions. Following functions and constants can be used in these expression:

- Functions

  sin,cos,tan,asin,acos,atan,atan2,sinh,cosh,tanh,
  exp,ln,log,flog10,sqrt,fabs,floor,ceil,rint

- Constants

  1. `PI`: value of π
  2. `C`: speed of light in meters per second.
  3. `R2D,D2R`: multiplicative constants for conversion from Radians to Deg. and vice versa
  4. `H2R,R2H`: multiplicative constants for conversion from Hours to Radians and vice versa
  5. `SOL2SID,SID2SOL`: multiplicative constants for conversion from Solar to Sidreal time and vice versa

Numbers in the expression can be in any of the following representations:

- Integer format, real (float) format
- 1.0E-1 or 1.0e-1 (=0.1)
- 1.0D-1 or 1.0d-1 (=0.1)
- 1h10m0.1s: the time format - converts the number to seconds before using it.
- $1d1'1''$: The angular format - converts to arc seconds before using it.

  ($1d1' = 3660.0''$ and $-1d1' = -3660.0''$)

A.1.1 De-referencing mechanism

The user interface system allows the application programmers to define shell symbols which are treated as constants (i.e. their values cannot be modified by the user). These symbols are referred to as the `const symbols`. Some applications may load a number of frequently used values of various keywords as `const symbols` (e.g., the list of symbols reported by the `showfmt` command of the `xtract`[1] program). However, by default, none of the `const symbols` are presented to the user and its the responsibility of application program to add extra shell-commands for viewing the list of `const symbols`.

[1] http://langur.ncra.tifr.res.in/~sanjay/Offline/xtract

Values can be transfered from one keyword or a `const symbol` to another keyword. E.g. to transfer the i^{th} value from keyword `Key1` to a keyword `Key2`, one could use `Key2 = $Key1[i]`. `Key2=$Key1` will transfer all values of `Key1` to `Key2`.

A.1.2 File name convention

Most off-line applications can perform I/O using UNIX pipes. If the input file name begins with '`<`', the rest of the file name is treated as a command, the output of which becomes the input of the application. For example, to supply the output of a program `tmac` as the input of the program `xtract`, the ``` ``in'' ``` keyword of `xtract` must be set to '`<tmac`". After the shell-command `go` is issued for `xtract`, the interface of `tmac` will be started and the keywords of `tmac` can then be set in the normal fashion. When the shell-command `go` is issued again, `tmac` will begin execution, and the output of `tmac` becomes the input of `xtract`. Till this time, execution of `xtract` would remain blocked, waiting for input from `tmac`.

Similarly, if the output file name begins with the '`|`' character, the rest of the file name is treated as a command, the input of which becomes the output of the off-line program.

By convention, a blank output file name implies that the output would go to the standard output stream (usually your screen) and a blank input file name implies that the input would be read from the standard input stream (usually your keyboard).

A.2 Customization

The user interface can be customized using the following environment variables.

- `GDOC`

  `GDOC` must point to the standard directory where documents explaining the various off-line data analysis programs are kept. The `explain` command will first look for the explanation file in the local directory and then in the directory specified by this variable. The name of the explanation file is constructed by appending the suffix ``` ``.doc'' ``` to the application name.

- `GERR, GOUT`

  `GERR` and `GOUT` variables are used when the application is run using the `gob` command (see section A.1). The standard output of the application will be redirected to the file specified by `GOUT` while the standard error stream will be redirected to the file specified by `GERR`. By default, these variables are set to `/dev/null`.

- `GCONF`

  Some applications may load frequently used setting for some keywords. These values are loaded as constants of the interactive shell and can be stored in a configuration file. Their values can be transfered to the application keywords by referring to their value by '$' mechanism (see section A.1.1). The path of the directory containing these configuration files is specified by the environment variable `GCONF`. The configuration filename is constructed by appending the suffix "`.config`" to the application name. If `GCONF` is not defined, the application looks in the directory specified by `GDOC`. If this variable is also not defined, or the configuration file is not found, the application will look for the configuration file in the current directory.

- `EDITOR`

  `EDITOR` environment variable is used to specify the name of the text editor to be used in the `edit` command (see section A.1). The default editor is `emacs`.

- `GDEFAULTS`

  The default values of keywords can be saved in "defaults file". `GDEFAULTS` variable specifies the directory where these files are stored. Such a file can be automatically loaded by the applications upon startup. The defaults file name is constructed by appending the suffix "`.def`" to the application name.

  By default, the application will look for the defaults file in the current directory. If a keyword appears in the "`.def`" as well as in the "`.config`" file, the keyword will be treated as a shell constant. This can be used to effectively produce specialized versions of an application program by writing an appropriated "`.config`", where keywords can have fixed values, not alterable by the user (e.g., a version of `xtract` which will read input from the shared memory of the GMRT data acquisition system).

The following two variables are effective only for versions of the user interface libraries which use the GNU Readline and the History libraries.

- `GHIST`

`GHIST` specifies the file in which the history of the commands issued in the interactive session is saved. This file will be common to all applications. The default history file is `$HOME/.g_hist`.

This is also the file from which all applications will load the command history.

- `MAXGHIST`

  `MAXGHIST` should be set to the maximum number of command history entries which the user wishes to save. By default this is set to 100.

A.2.1 The `help` keyword

All applications are sensitive to the keyword `help` (not to be confused with the shell command `help` described in section A.1). It is never displayed in the list of keywords. To use it, it must be specified as a command-line argument. Following is the list of values that can be set for this keyword on the command-line and their effect:

- `help=noprompt`

  The application runs in the non-interactive mode. This is useful when the application is run from a shell script.

  When run in this mode, value of all the keywords which needs to be set must be supplied on the command-line (the order of the command-line options is not important). The keywords must be fully spelled in the command-line options (i.e., no minimum-match will be applicable).

- `help=explain`

  This executes the `explain` command of the interactive shell (see section A.1) without starting the interactive shell.

  If a keyword is supplied within brackets (`'('` and `')'`) immediately after the `explain` string (e.g. `help=explain(out)`, help will be provided for the specified keyword alone. If an application name is also included within the brackets, separated from the keyword by a colon (`':'`) (e.g. `help=explain(out:xtract)`, help for the named keyword of the named application will be provided. If the keyword is skipped (but not the colon), entire help of the named application will be provided.

- `help=doc`

This results into an empty documentation file written on the standard output, in the required format, with a list of keywords to which the application is sensitive. This is for use by authors of the applications and to encourage them to not only write the documentation, but also in a uniform format.

- `help=dbg`

  Application programmers can define hidden keywords which are normally not displayed for the user, but will be used by the application internally. These keywords can be accessed as normal keywords by setting `help` keyword to the value "`dbg`".

Appendix B

`xtract`: Usage and programming with the library

This appendix describes the design of the program `xtract` and the macro language used to extract visibility and auxiliary data from GMRT LTA database. Section A.1 is intended for "plain" users of the program. The `xtract` macro language parser is also available as a stand-alone library, which can be used in other applications. The internal design of the library is described in section B.3 and is targeted for more enterprising users, who will find if useful to extend this program. Section B.4 describes the application programmers interface (API) of the library. Section B.5 describes the mechanism to extend the list of data/parameters which can be extracted.

B.1 The macro language

The macro language encapsulates the fact that whatever numbers we need to extract from the visibility database are either antenna based, interferometer based (or equivalently, baseline based) and/or a function of time and/or frequency within the given observing band. One may want to extract data for a list of antennas, baselines, frequency channels with selection applied in time.

The macro language syntax is a hybrid of the implicit loops of the `write` statement of `FORTRAN` (see the manual for `FORTRAN`) and the format string used by the output functions of `C` (see documentation on `printf` in the manual for `C`). Three operators are defined in the language, namely `base`, `chan` and `ant`, which loop over a list of baselines, frequency channels and antennas respectively. The implicit loops loop over the *body* of the operators. The *body* is a list of semicolon (';') separated list of *elements* enclosed in pair of curly braces ('{' and '}'). Each operator must be followed by a *body* and the *elements* of the *body* can be another operator-*body* pair. Hence, nested loops are possible.

In the `xtract` program, the list of values for the operators is supplied as a list of comma (',') separated values for the keywords `baselines`, `channels`, and `antenna` respectively. An operator for time range selection is also required, but not explicitly defined. All macros are internally the *body* of this time range operator. The time range selection can be specified via the `timestamps` keyword.

Various *elements* which the syntax recognizes are listed in Table B.1. Some of these are independent of any operator, while others need to be part of the *body* of one or more operators. The *elements* and the operators required by these *elements* are also listed in the table.

Table B.1: List of *elements* which can be part of the *body* of operators in `xtract` macros

Elements	Operators Required	Description
`ua,va,wa`	`ant` (antenna based)	(u,v,w) co-ordinates of the antennas
`u,v,w`	`base` (baseline based)	u_{ij} (=$\mathtt{ua}_i - \mathtt{ua}_j$) for the baseline ij
`re,im`	`base,chan`	Real and Imaginary parts of the visibility
`a,p`	`base,chan`	Amplitude and phase of the visibility
`cno`	`chan`	Frequency channel number
`ha,ist,lst`	none	Hour Angle, IST time stamp and LST in hrs.
`az, el`	none	Antenna Azimuth and Elevation angles in degree
`delay,phs0, dphs`	`ant`	The delay in μs applied to the antennas at the delay cards, the fringe rotation phase, and the phase ramp applied at the output of FFT

`ua,va,wa` are the co-ordinates of the antenna in the (u, v, w) co-ordinate system in units of the wavelength of the center of the observing band and `u=ua`$_1$`-ua`$_2$, where the subscripts refer to the two antennas of a baseline.

The macro to produce a table of rows with Hour Angle (HA) value in the first column followed by two columns for the real and imaginary parts of the visibility at a single frequency for all selected baselines, would be

```
fmt=base{ha;chan{re;im};\\n}
```

The special element \\n represents the actual character that will appear at the given position in the output (which is the `NEWLINE` character here). The only other special element that this syntax currently allows is \\t (TAB).

The (*u,v,w*) values for each baseline can be added to each row of the table by the following macro

```
fmt=base{ha;u;v;w;chan{re;im};\\n}
```

However note that the following macro is in error

```
fmt=ha;u;v;w;base{chan{re;im};\\n}
```

This is because the *elements* `u,v,w` are a function of the baseline and they do not appear as part of the *body* of the `base` operator. The macro parser will generate an error message pointing out the possible error in this macro.

The *elements* can also be qualified by a `C`-styled `printf` format field. Hence, for example, if the value HA needs to be written with field length of 8 characters and precision of 3 digits, the `fmt` string would become

```
fmt=base{ha%8.3f;u;v;w;chan{re;im};\\n}
```

The format for the numbers can be of type 'f','g','G','e', or 'E' (see the documentation on `printf` function of `C` language for more details).

Various output formats can be generated by changing the order of loops and *elements* in this syntax. Here are some examples. Each of these will generate a table. The values in the various columns will be as given in the explanation.

- `fmt=base{ha;u;v;w;chan{re;im};\\n}`

  Column 1 will be the Hour Angle. Columns 2,3, and 4 will have the *u,v,w* values followed by $2 \times N$ columns for real and imaginary values for the N values that the `chan` operator can take. There will be one such row in the table for each value of the `base` operator.

- `fmt=ha;lst;\\n;base{u;v;w;chan{re;im};chan{a;p};\\n}`

  This format will generate a table with rows of unequal lengths.

  Row 1 will have only HA and LST values.

  Row 2 will have u,v,w in the first 3 columns followed by real, imaginary, amplitude and phase for all channels listed in the `chan` operator. There will be one such row for every value of the `base` operator.

- `fmt=ha;base{u;v;w};\\n;base{chan{re;im}};\\n`

  This macro will generate a table of set of two rows of unequal lengths per input data record.

  First row will have the HA and u, v, w values for each selected baseline.

  Second row will have the real and imaginary values of the visibilities for all channels of the `chan` operator and for all values of the `base` operator.

B.2 Output filters

The macro language is used by the application program `xtract` to extract data from the GMRT visibility database. Most common use of `xtract` is to extract a data in the form of an ASCII table for display and/or further processing (e.g., to compute the antenna pointing errors). The output of `xtract` can be supplied to another program in two ways.

By default, `xtract` writes the output on the standard output. Hence if `xtract` is started as

```
xtract | MyProg
```

the output of `xtract` will be piped to the standard input of the program named `MyProg`. The other, probably more convenient, method of piping data is to set the `out` keyword to '`|MyProg`'.

The output will be written in ASCII format, preceded by a simple header. Apart from other fields, the header contains information about the number of rows and columns and the labels for each of the columns. This header always ends with a string "`#End`", after which the data is written. A line beginning with '`#`' is also written per LTA-scan. It is hoped that users will utilize these facilities to generate more filters to process and display data externally.

If the output file name begins with a '*', the file name is constructed after stripping the initial '*' character and the data is written in binary format (floating point numbers of size determined by the operator `sizeof(float)` of `C` or `C++`). The data itself is preceded by the ASCII header mentioned above. Hence, `out=*tst.bin` will produce a file `tst.bin`, which will contain the output in binary format and `out=*|MyProg` will pipe the binary data to `MyProg`.

For convenience of usage, a filter has been incorporated on the output stream of `xtract` which will supply the data directly to the `QDP` line plotting package. This filter can be invoked by setting `out=>QDP`. The output, in this case will be displayed as a stack of line plots using `QDP`.

A more general and usable graphical interface to the multiplot features of the freely available line plotting program `Gnuplot` has been developed by (Kudale & Bhatnagar NCRA Tech. Rep. - in prepration). The data to this software can be supplied using the piping mechanism described above. A graphical user interface then allows the user to select the available baselines/antennas and plot them interactively in a flexible manner.

B.3 Internal design

The `xtract` macros are first interpreted and then compiled in the memory. This complied code is then executed for every input data record. The details about the compilation and execution of the format string are given below.

B.3.1 Macro compilation

The process of compilation of the format string involves two steps.

First, all the loops represented by the operators in the macro are exploded into a linked list (also called the symbol table), with each node of the list corresponding to a valid *element* of the language. Each *element* is represented in the memory by a structure of the following type:

```
typedef struct StructSymbType {
  char Name[NAMELEN];
  char Fmt[FMTLEN];
  int abc[3];
  unsigned int Type;
```

```
    float (*func)(char *,float **,int);
    float *fargv[NARGV];
    int fargc;
    float *ptr;
    struct StructSymbType *next;
  } SymbType;
```

All recognized *elements* (symbols) are tabulated in the memory in a temporary table, which is a list of structure of the following type:

```
  typedef struct TT {
    char *Name;
    unsigned int Class,Type;
  } TypeTable;
```

This table is hard-coded in the file `table.h` and is used only to validate the symbols in the macro. Once validated, the `Class` and `Type` information for this table is transfered to the actual symbol table and the temporary table destroyed.

Apart from the name of the *element* and the `C` styled format string, the nodes of the symbol table also have information about the mechanism to get the numeric value associated with the *element*. This information is in the field `Type` of the structure above. Valid types for the *elements* are listed in Table B.2.

Table B.2: Table of element types in `xtract` macro language

Type	Meaning
CHARType	Represents a character to output
FTYPE	Function type: the value will be returned by a call to the function `func`
PTYPE	Pointer type: the value will be in the buffer at the location pointed to by `ptr`

The `abc` field of the *element* structure shown earlier, holds the values of the three operators (`ant`,`base`, and `chan`) applicable to the *element*.

Before an element is added to the symbol table, a check is made to ensure that all the required operators (listed in Table B.1) are active. To generate this information about the required operators, *elements* are further categorized into one of the classes listed in Table B.3.

Table B.3: Table of valid classes of the *elements* of `xtract` macro language

Class	Operators Required
IV	None
AV	`ant`
BV	`base`
CV	`chan`
BCV	`base,chan`
ABCV	`ant,base,chan`

Once the element is validated for the required active operators, a new link is created in the symbol table and filled with the `Name`, `Type` and `Class` of the *element*. By this time, the loops (represented by the list of values associated with various operators) have already been exploded (i.e., a node created in the symbol table for each value of the operator). Information about the values of the operators is transfered to the symbol table for every value of the active operators and the values of the required operators are put in the `abc` array (passive operators are assigned a value of -1). By this time, if no error has occurred, it is assured that the syntax was correct and all the *elements* in the macro were recognized.

Second step in the process of compilation is to fill in the information about the mechanism to get the numeric values of each *elements* in the list. The `Type` of the element and, if required, the values in the `abc` array are used for filling in this information.

For *elements* of type `PTYPE` , the `ptr` field is made to point to the location in the memory where the required value is to be found. This type of *element* refer to particular values in the buffer in the memory and need the offsets in the buffer which can be computed using the `abc` array. The buffer in the memory is generally the buffer in which data records from the LTA-file is read. Examples of this kind of *elements* are IST, real/imaginary values of the visibility, etc.

For *elements* of type `FTYPE`, the `func` field is filled with a pointer to a function which will be called when the value of the element is required. If the computation of the value requires some data, the pointers to this data is put in the field `fargv` and the total number of such pointer is put in the field `fargc`. These will be passed as arguments to the function when the value of the element is required. The first argument passed to the function will be the name of the element. Examples of this kind of *elements* are HA, amplitude/phase of the visibility, etc.

For *elements* of type `CHARType`, nothing needs to be done. The name of such *elements* is the character that is to be copied to the output during execution.

B.3.2 Macro execution

The process of "execution" of the compiled list of *elements* is rather simple. The program steps through the entire list of *elements* and checks the type of each element on the list. If the type is `PTYPE`, the value of memory location to which `ptr` points, is copied to the output stream using the format in the `Fmt` field of the *element*. If the type is `FTYPE`, the function specified by `func` is called with `Name`, `fargv`, and `fargc` as the arguments. The value returned by this function is then copied to the output stream using the format in the `Fmt` field of the element. If the type is `CHARType`, the first character of the `Name` field is copied to the output stream.

Following is an example of a simple routine used for execution of the compiled macros:

```
/* $Id: xtract.tex,v 1.9 2000/02/18 03:58:24 sanjay Exp sanjay $ */
#include <stdio.h>
#include <fmt.h>

int ExecuteDef(FILE *fd,SymbType *P,float *buf,int len)
{
  SymbType *i;
  int N=0;

  for (i=P;i;i=i->next)
    {
     switch(i->Type)
       {
        case PType:
         {buf[N++]= *i->ptr;break;}
        case FType:
         {buf[N++]=i->func(i->Name,i->fargv,i->fargc);break;}
```

```
        case CHARType: return N;
        default:
          fprintf(stderr,"###Error: Unknown type in ExecuteDef\n");
       }
    }
  return N;
}
```

If the output in required is the binary format, one can write an equivalent `Execute` routine, which will ignore the `Fmt` field and `CHARType` *elements* and output the values in the binary format.

B.4 Programming with the `xtract` library

The process of compilation and execution of the `xtract` macro described above is done via a stand-alone library. This section describes the Application Programming Interface (API) of this library.

The `C/C++` interface of this library is defined in `fmt.h`, which must be included in the code and linked to `libjump.a`, in addition to all other GMRT Off-line libraries (`liboff.a, libregex.a, libkum.a`).

B.4.1 Interpretation and compilation of the format string

The `xtract` macros are interpreted via the following function call:

```
int interpret(char *fmtString, struct fftmac *fm,
              Parameters *Params, SymbType *Inst)
```

The first argument is the macro as a NULL terminated string. Second argument is the `fftmac` (Bhatnagar 1997a)[1] structure which holds the various mappings for the *LTA* database (e.g., sampler to the *MAC* mapping, etc.). This structure must be filled using services provided by the GMRT Offline Library[2] (`getFFTMac method`). The third argument is a pointer to the structure of the type `Parameters`. This structure holds the various parameters which the library uses while executing the macro. Various fields of this structure are described in Section B.4.3. The value of some of the fields of this structure are defined by

[1]http://langur.ncra.tifr.res.in/~sanjay/Offline/gstruct
[2]http://langur.ncra.tifr.res.in/~sanjay/Offline/ltaobj

the user, while others are to be extracted from the LTA database. The fourth argument is a pointer to a structure of type `SymbType`. This is the table of *elements* mentioned earlier and must be initialized to NULL before being passed to this routine. A return value of less than `EOF` (-1), indicates a syntax error in the macro.

Compilation of the macro string is done via a call to:

```
int Compile(SymbType *Inst,struct fftmac *fm,
            struct AntCoord *Tab, Parameters *Params)
```

The first argument is the symbol table returned by a call to `interpret`. It now points to the head of a linked list of nodes of type `SymbType`. The last node of this list is NULL. The second argument is the `fftmac` structure. The third argument is the table of antenna co-ordinates. This can be retrieved from the LTA database via the services provided by the GMRT Offline Library[3] (`getFFTMac method`). The fourth argument is a pointer to the `Parameters` structure. A return value of less than `EOF`(-1) indicates error in compilation of the macro. On successful compilation, it returns the size of the compiled symbol table in units of the size of the structure `SymbType`.

B.4.2 Execution of the compiled macro

If the interpretation and compilation of the macro was successful, the compiled macro can be executed via calls to a user-supplied function of signature

`int Exec(FILE *fd, SymbType *Inst, float *Buf, int ProgSize)` `fd` refers to the output file already opened for writing. `Inst` is the symbol table returned by `interpret`. In case the output data is not to be written to any file, the user can write versions of this routine which will fill the data in the buffer `Buf`. `ProgSize` is the value returned by `Compile`.

The `data` field of the `Parameters` structure (see section B.4.3) must be made to point to the buffer in which the LTA-data buffers are read. To generate a regular stream of output, corresponding to each input data record, this function must be called every time a new LTA-data record is read.

Few types of `Exec` functions are provided in the library. These include:

- `Execute`

  It writes the output data to the `fd` file descriptor. It does not use the `Buf` pointer.

[3] http://langur.ncra.tifr.res.in/~sanjay/Offline/ltaobj

- `ExecuteDef`

  This writes the output data to the buffer pointed by `Buf`. The size of the this buffer must be big enough to hold one floating point number per node of the symbol table (return value of `Compile`). This does not use the file descriptor.

- `ExecuteQDP`

  This supplies output data to the `QDP` program via a pipe opened via the `libpipe.a` library. This uses the `Buf` pointer but does not use `fd`.

To generate any other functionality, the programmers need to write their versions of this function. The recommended route for writing a new function is to modify `Execute` or `ExecuteDef` functions.

B.4.3 The `Parameters` structure

The `Parameters` structure is of the following type:

```
typedef struct StructParamType{
   int Norm;
   int *BList, *SList, *AList, *CList;
   int  NBase,  NScans, NAnt,   NChan;
   int dBNBase, dBNChan, dBNAnt;
   int dBStartChan;
   int TimeOff, ParOff,DataOff;

   float Lambda;
   float sd,cd, TUnits;
   char *data;
} Parameters;
```

Various fields and their use is as follows:

- *int Norm*

  This must be set to 1 if the visibility data is to be normalized by the geometric mean of the self correlations. Otherwise this must be set to 0.

- `int *BList, *SList, *AList, *CList`

These are pointers to the user selected list of the baseline, scan, antenna and channel numbers respectively. The list of channel numbers must be 0-relative and not the absolute channel index of the data base (which could start with number between 0 to maximum number of channels).

Typically, the user selects the baselines and antennas via the baseline/antenna names. These are supplied as strings by the user. Two functions, `MkBaselines` and `MkAntNo`, are provided to convert these stings to a list of bit fields in which the bits corresponding to the selected baselines are set to 1. Another routine `toIntList`, is provided to convert the bit fields to list of integers representing the selected baselines. These functions are available in the library `liboff.a` and are described in Appendix A of GMRT Offline Library[4].

- `int NBase, NScans, NAnt, NChan`

  These are the lengths of `BList, SList, AList` and `CList`.

- `int dBNBase, dBNChan, dBNAnt`

  These are the number of the baselines, channels and antenna in the data base.

- `int dBStartChan`

  This is the number of the first frequency channel in the data base.

- `int TimeOff, ParOff, DataOff`

  These are the offsets within the LTA-data buffer to locate the time stamp, the auxiliary parameters, and the visibility data itself. These offsets can be extracted from the *global header* of the database.

- `float sd,cd`

  These are the values of $\sin(\delta)$ and $\cos(\delta)$ where δ is the declination of the pointing center of the telescope. These are used for the calculation of the (u, v, w) coordinates during execution.

- `float TUnits`

  This is the multiplication factor used to convert the time stamp in the data to seconds of time. This is also extracted from the *global header*.

[4]http://langur.ncra.tifr.res.in/~sanjay/Offline/ltaobj

- `float Lambda`

  This the wavelength of the observing frequency in meters.

- `char *data`

  This is the pointer to the beginning of the LTA-data buffer.

B.5 Adding new *elements* to the syntax

To add a new *elements* to the `xtract` macro language, one needs to define the values of `Name`, `Class`, and `Type` of the new symbol in the table of valid *elements*. This is done by adding to the table in the file `table.h` (make sure the last element of this list is left unaltered).

One also needs to add a piece of `C`-code, which will fill the required fields of the structure `SymbType` (depending upon the `Type` of the element – the `ptr` field for `PTYPE` *elements* and the `func, fargv,` and `fargc` fields for `FTYPE` *elements*). It is the responsibility of the programmer to make sure that this code is correct in terms of getting the numeric value of the *elements*. Also, the programmer must make sure that this code is compatible with the `Type` of the *element*. Failing to do so will either generate wrong values or crash the program at the time of execution. This code is to be added in the function `Compile` in the file `Compile.c`. The application will need to be rebuilt for the new symbol to be recognized in the `fmt` syntax.

Appendix C

Antenna/baseline naming convention

C.1 Antenna naming convention

An antenna can be specified by the index of the antenna or a regular expression. A full antenna name consists of three hyphen ('`-`') separated fields. The first field is the antenna name, the second is the side band name and the third is the polarization name. Each of the three fields can be a regular expression. All missing fields after the last specified field are replaced by "wild card" regular expressions.

E.g., a fully qualified name for `C11`, upper side band, 175-MHz polarization channel (nominally the LL polarization) can be expressed as:

`C11-USB-175`

All central square antennas, with upper side band and 130-MHz (nominally the RR polarization) polarization channel can be expressed as:

`C.+-USB-130`

(Here '.' matches one instance of any character and the '+' operator operates on '.' zero-or-more number of times. Hence ".+" is equivalent to the '*' wild card character. This is the POSIX regular expression syntax).

All central square antennas with upper sideband and either polarization channel can be expressed as:

`C.+-USB-.+`

or simply

`C.+-USB`

Finally, all central square antennas without any selection on the side band and polarization channel can be expressed as:

`C.+-.+-.+`

or simply

`C.+`

All above examples, but with only arm antennas can be expressed by replacing "`C.+`" in the above examples by "`[EWS].+`".

C.2 Baseline naming convention

Baselines can be specified by the index of the baseline or a regular expression describing a baseline name.

A full baseline name is composed of two antenna names separated by a colon (':'). A fully qualified baseline name between the upper-side band, 175-MHz polarization signals from antennas `C11` and `C12` would be

`C11-USB-175:C12-USB-175`

Any of these fields can be regular expressions. Hence to choose all baselines with antenna `C11` upper side-band signals from either of the polarization, one would use the baseline name as:

`C11-USB-.+:.+`

If only the first field of antenna name is given, the others are taken to be the wild card characters.

If second fully qualified antenna names is missing from the baseline name, it is also replaced by wild card.

These selections will exclude all self correlations. Also, if a baseline has already got selected in a previous selection, it will be excluded from all later selections.

To select the self correlations, one must add 'A' to be beginning of the antenna names. In such a case, the name of the second antenna is redundant and therefore not required.

C.2.1 Examples

1. Baseline name to select all baselines with with `C11` = `C11`

2. Baseline name to select the self correlation of `C11` = `C11:C11` or `AC11`

3. Baseline name select the self correlation of `C11`, 130-MHz polarization channel from both side bands = `AC11-.+-130`

4. Baseline name to select *all* self correlations = `A.`

5. Baseline name to select *all* lower side-band baselines with `C11` = `C11-USB-.+:.` or `C11-USB-.+.`

6. Baseline name to select *all baselines in the database* = `.`

7. Baseline name to select *all* central square baselines with `C02-USB-175` = `C02-USB-175:C.+`

8. Baseline name to select *all* baselines of `C02-USB-175` with only arm antennas = `C02-USB-175:[EWS].+`

Appendix D

Computation of antenna based complex gains

The normalized cross-correlation function (the correlator output), measured by an interferometer using two antennas labeled by i and j, in the limit $I \ll T_i^s/\eta_i$, can be written as:

$$\rho_{ij}^{Obs} = \rho^{Obs}(u_{ij}, v_{ij}, w_{ij}) = \iint\limits_{-\infty}^{+\infty} I(l,m)\sqrt{\frac{\eta_i\eta_j}{T_i^s T_j^s}}\, e^{2\pi\iota(u_{ij}l + v_{ij}m + w_{ij}\sqrt{1-l^2-m^2} + \phi_i - \phi_j)} \frac{dl\; dm}{\sqrt{(1-l^2-m^2)}} + \epsilon_{ij} \tag{D.1}$$

where $I(l,m)$ is the sky surface brightness, η_i is the sensitivity and T_i^s is the system temperature of the antenna i in units of Kelvin/Jy and Kelvin respectively, ϵ_{ij} is the additive noise on the baseline i-j, and ϕ_i is the antenna based phase of the signal. The rest of the symbols have the usual meaning.

In practice, however, the antenna based amplitude ($\sqrt{\eta_i/T_i^s}$) and phase (ϕ_i) are potentially time varying quantities. This could be due to changes in the ionosphere, temperature variations, ground pick up, antenna blockage, noise pick up by various electronic components, background temperature, etc. Treating the quantities under the square root in the above equation as the antenna dependent amplitude gains, these can be written as complex gains $g_i = a_i e^{-\iota\phi_i}$ where $a_i = \sqrt{\eta_i/T_i^s}$. For an unresolved source at the phase tracking center, variations in this amplitude will be indistinguishable from a variations in the ratio of η and T^s.

In terms of g_is, we can write Equation D.1 as

$$\rho_{ij}^{Obs} = g_i g_j^{\star} \rho_{ij}^{\circ} + \epsilon_{ij} \tag{D.2}$$

where

$$\rho_{ij}^{\circ} = \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} I(l,m)\, e^{2\pi\iota(u_{ij}l + v_{ij}m + w_{ij}\sqrt{1-l^2-m^2})} \frac{dl\ dm}{\sqrt{(1-l^2-m^2)}} \tag{D.3}$$

The use of the word “antenna based gains” for g_is result in confusion for many and needs some clarifications. g_is are called complex “gains” since they multiply with the complex quantity ρ_{ij}. For an unresolved source, $|g_i|$ represents the fraction of correlated signal and $arg(g_i)$ represents the phase of the correlated part of the signal from the antenna with respect to the phase reference (usually the reference antenna). It is in this sense that it is referred to as “antenna based” gains. However, as defined here, they include T^s which in turn includes the sky background temperature. They are therefore a function of direction in the sky. However, here we assume that the angular scale over which g_is vary is larger than the antenna primary beam (isoplanatic case).

For an unresolved source at the phase tracking center, all terms in the exponent of ρ_{ij}° are exactly zero. ρ_{ij}° in this case would be proportional to the flux density of the source.

Assuming that the antenna dependent complex gains are independent, with a gaussian probability density function (this implies that the real and imaginary parts are independently gaussian random processes), one can estimate g_is by minimizing, with respect to g_is, the function S given by

$$S = \sum_{\substack{i,j \\ i\neq j}} \left|\rho_{ij}^{Obs} - g_i\ g_j^{\star}\ \rho_{ij}^{\circ}\right|^2 w_{ij} \tag{D.4}$$

where $w_{ij} = 1/\sigma_{ij}^2$, σ_{ij} being the variance on the measurement of ρ_{ij}^{Obs}

Dividing the above equation by ρ_{ij}° (the source model, which is presumed to be known – it is trivially known for an unresolved source), and writing $\rho_{ij}^{Obs}/\rho_{ij}^{\circ} = X_{ij}$, we get

$$S = \sum_{\substack{i,j \\ i\neq j}} \left|X_{ij} - g_i\ g_j^{\star}\right|^2 w_{ij} \tag{D.5}$$

If ρ_{ij}° represents the structure of the source accurately, X_{ij} will have no source dependent terms and is purely a product of the two antenna dependent complex gains.

Expanding Equation D.5, we get

$$S = \sum_{\substack{i,j \\ i \neq j}} \Big[\; |X_{ij}|^2 - g_i^\star g_j X_{ij} - g_i g_j^\star X_{ij}^\star + g_i g_i^\star g_j g_j^\star \Big] \; w_{ij} \tag{D.6}$$

Evaluation $\frac{\partial S}{\partial g_i^\star}$ and equating it to zero [1], we get

$$\frac{\partial S}{\partial g_i^\star} \;=\; \sum_{\substack{j \\ j \neq i}} \left[-g_j X_{ij} w_{ij} + g_i g_j g_j^\star w_{ij} \right] \;=\; 0 \tag{D.7}$$

or

$$g_i \;=\; \frac{\sum\limits_{\substack{j \\ j \neq i}} \; X_{ij} g_j w_{ij}}{\sum\limits_{\substack{j \\ j \neq i}} \; |g_j|^2 \, w_{ij}} \tag{D.8}$$

This can also be derived by equating the partial derivatives of S with respect to real and imaginary parts of g_i as shown in Section D.3.

Since the antenna dependent complex gains also appear on the right-hand side of Equation D.8, it has to be solved iteratively starting with some initial guess for g_js or initializing them all to 1.

Equation D.8 can be written in the iterative form as:

$$g_i^n \;=\; g_i^{n-1} + \alpha \left[g_i^{n-1} - \frac{\sum\limits_{\substack{j \\ j \neq i}} \; X_{ij} g_j^{n-1} w_{ij}}{\sum\limits_{\substack{j \\ j \neq i}} \; \left| g_j^{n-1} \right|^2 w_{ij}} \right] \tag{D.9}$$

where n is the iteration number and $0 < \alpha < 1$. Convergence would be defined by the constraint

$$|S_n - S_{n-1}| < \delta \tag{D.10}$$

(the change in S from one iteration to another) where δ is the tolerance limit.

[1] Complex derivatives can be evaluated by treating g_i and $g_i^\star$ as independent variables. See (Palka 1990)

D.1 Interpretation of the equation

Equation D.8 offers itself for some intuitive understanding in the following way.

X_{ij} is a product of two complex numbers, namely g_i and $g_j^\star$, which we wish to determine. X_{ij} is itself derived from the measured quantity V_{ij}^{Obs}. Numerically speaking, each term in the summation of the numerator of Equation D.8 will involve g_i (via X_{ij}) and the multiplication of X_{ij} with $g_j w_{ij}$ would give g_i an effective weight of $|g_j|^2 w_{ij}$. Since the denominator is the sum of this effective weight, the right-hand side of Equation D.8 can be interpreted as the weighted average of g_i over all correlations with antenna i.

In the very first iteration, when $g_j = (1, 0)$, the normalization would be incorrect since the numeric value of g_j, as it appears inside X_{ij} would be different from that used in the denominator of Equation D.8. However, as the estimates of g_js improve with iterations, the equation would progressively approach a true weighted average equation. The speed of convergence will depend upon the value of α and the convergence would be defined by the constraint in Equation D.10. In the ideal case when the true value of all g_is is known, right hand side of Equation D.8 also reduces of g_i.

Estimating g_i for an antenna, by averaging over the measurements from all baselines in which it participates (for a unresolved source) makes sense since for an N element array, g_i would be present in N-1 measurements (all the $X_{ij}|_{j=1,N;j\neq i}$) and the best estimate of g_i would be the weighted average of all these measurements. Proper weight for g_i, buried in each of the products X_{ij}, can be found heuristically as follows. g_i, estimated from the measurements of a given baseline, must obviously be weighted by the signal-to-noise ratio on that baseline. This is w_{ij} in the above equations. It must also be weighted by the amplitude gain of the other antenna making the baseline, namely g_j, to account for variation in antenna sensitivities and T^s. The total weight for g_i would then be $|g_j|^2 w_{ij}$, the sum of which appears in the denominator of Equation D.8. Knowing that ideally $X_{ij} = g_i g_j^\star$, each of the $X_{ij}|_{j=1,N}$ must be multiplied by $g_j w_{ij}$ (to apply the the above mentioned weights to g_i), before being summed for all values of j and normalized by the sum of weights to form the weighted average of g_i. One thus arrives at Equation D.8 using these heuristic arguments.

D.2 Estimation of the system temperature (T^s)

For an unresolved source of known brightness I, in the limit $T^a \ll T^s$, $\rho_{ij}^\circ = I$ and Equation D.1 can be written as

$$\rho_{ij}^{Obs} = I g_i g_j^{\star} \approx I \sqrt{\frac{\eta_i \eta_j}{T_i^s T_j^s}} \tag{D.11}$$

where $\eta_i = A_e/2k_b$, A_e is the effective area of the dish, k_b is the Boltzman's constant and

$$|g_i| = \sqrt{\frac{\eta_i}{T_i^s}} \tag{D.12}$$

Hence, knowing η_i, T_i^s can be estimated from the amplitude of the antenna dependent complex gains.

All contributions to ρ_{ij}^{Obs}, which cannot be factored into antenna dependent gains, will result in the reduction of $|g|$. η remaining constant, this will be indistinguishable from an increase in the effective system temperature. Since majority of later processing of interferometry data for mapping (primary calibration, bandpass calibration, SelfCal, etc.) is done by treating the visibility as a product of two antenna based numbers, this is the effective system temperature which will determine the noise in the final map (though, as a final step in the mapping process, baseline based calibration can possibly improve the noise in the map).

In the normal case of no significant baseline based terms (ϵ_{ij}) in X_{ij}, the system temperature as measured by the above method will be equivalent to any other determination of T_i^s.

T^s can also be determined by recording interferometric data for a strong point source with and without an independent noise source of known temperature at each antenna. In this case

$$T_i^s = T_i^n \left(\frac{g_i^{ON^2}}{g_i^{OFF^2} - g_i^{ON^2}}\right) \tag{D.13}$$

where g_i^{ON} and g_i^{OFF} are the antenna dependent gains with and without the noise source of temperature T^n. Note that η_i does not enter this equation. Also, T^n should be such that $\sqrt{T^a/(T^n + T^s)} \geq 0.1$ to ensure that the correlated signal is measured with sufficient signal-to-noise ratio. For example, for P-band, a calibrator with P-band flux density > 5 Jy must be used.

D.3 Derivation of g_i using real and imaginary parts

g_is are complex functions. One can therefore write S in terms of g_i^I and g_i^p, the real and imaginary parts of g_i and minimize with respect to g_i^I and g_i^p separately. It is shown here that the complex arithmetic achieves exactly this and the results are same as that given by complex calculus. The superscripts I and R in the following are used to represent the real and imaginary parts of complex quantities.

Expanding Equation D.5, ignoring w_{ij}s and writing it in terms of real and imaginary parts we get

$$\begin{aligned}
\sum_{\substack{i,j\\i\neq j}} \left|X_{ij} - g_i g_j^\star\right|^2 &= \sum_{\substack{i,j\\i\neq j}} \left[X_{ij} - g_i g_j^\star\right]\left[X_{ij}^\star - g_i^\star g_j\right] \\
&= \sum_{\substack{i,j\\i\neq j}} \left[\left(X_{ij}^R + \iota X_{ij}^I\right) - \left(g_i^p + \iota g_i^I\right)\left(g_j^p - \iota g_j^I\right)\right] \\
&\qquad \left[\left(X_{ij}^R - \iota X_{ij}^I\right) - \left(g_i^p - \iota g_i^I\right)\left(g_j^p + \iota g_j^I\right)\right] \\
&= \sum_{\substack{i,j\\i\neq j}} \left[\left(X_{ij}^R - g_i^p g_j^p - g_i^I g_j^I\right) + \iota\left(X_{ij}^I + g_i^p g_j^I - g_i^I g_j^p\right)\right] \\
&\qquad \left[\left(X_{ij}^R - g_i^p g_j^p - g_i^I g_j^I\right) - \iota\left(X_{ij}^I + g_i^p g_j^I - g_i^I g_j^p\right)\right] \\
&= \sum_{\substack{i,j\\i\neq j}} S_0 S_0^\star
\end{aligned} \tag{D.14}$$

where

$$S_0 = \left[X_{ij}^R - g_i^p g_j^p - g_i^I g_j^I\right] + \iota\left[X_{ij}^I + g_i^p g_j^I - g_i^I g_j^p\right] \tag{D.15}$$

Taking partial derivative of S with respect to g_i^p and reintroducing w_{ij}, we get

$$
\begin{aligned}
\frac{\partial S}{\partial g_i^p} &= \sum_{\substack{j \\ j\neq i}} \left\{ \left[-g_j^p + \iota g_j^I \right] S_0^\star - S_0 \left[g_j^p + \iota g_j^I \right] \right\} w_{ij} \\
&= -\sum_{\substack{j \\ j\neq i}} \left[S_0 g_j + g_j^\star S_0^\star \right] w_{ij} \\
&= -2 \sum_{\substack{j \\ j\neq i}} Re\left(S_0 g_j w_{ij}\right) \\
&= -2 \sum_{\substack{j \\ j\neq i}} \left[\left(X_{ij}^R - g_i^p g_j^p - g_i^I g_j^I \right) g_j^p + \left(X_{ij}^I + g_i^p g_j^I - g_i^I g_j^p \right) g_j^I \right] w_{ij} \\
&= -2 \sum_{\substack{j \\ j\neq i}} \left[X_{ij}^R g_j^p - X_{ij}^I g_j^I - g_i^p {g_j^I}^2 - g_i^p {g_j^p}^2 \right] w_{ij}
\end{aligned}
\tag{D.16}
$$

Therefore,

$$
\frac{\partial S}{\partial g_i^p} = -2 \sum_{\substack{j \\ j\neq i}} \left[Re(X_{ij} g_j^\star) - \left|g_j\right|^2 g_i^p \right] w_{ij} \tag{D.17}
$$

Equating $\frac{\partial S}{\partial g_i^p}$ to zero, we get

$$
g_i^p = \frac{\sum_{\substack{j \\ j\neq i}} Re(X_{ij} g_j^\star w_{ij})}{\sum_{\substack{j \\ j\neq i}} \left|g_j\right|^2 w_{ij}} \tag{D.18}
$$

Similarly

$$
\frac{\partial S}{\partial g_i^I} = -2 \sum_{\substack{j \\ j\neq i}} \left[Im(X_{ij} g_j^\star) - \left|g_j\right|^2 g_i^I \right] w_{ij} \tag{D.19}
$$

Therefore the equivalent imaginary part of Equation D.18 is

$$
g_i^I = \frac{\sum_{\substack{j \\ j\neq i}} Im(X_{ij} g_j^\star w_{ij})}{\sum_{\substack{j \\ j\neq i}} \left|g_j\right|^2 w_{ij}} \tag{D.20}
$$

writing $g_i = g_i^p + \iota g_i^I$ and substituting for g_i^p and g_i^I from Equation D.18 and D.20 respectively, we get

$$g_i = \frac{\sum\limits_{\substack{j \\ j\neq i}} X_{ij} g_j^{\star} w_{ij}}{\sum\limits_{\substack{j \\ j\neq i}} |g_j|^2 w_{ij}} \tag{D.21}$$

This is same as Equation D.8, which was arrived at by evaluating a complex derivative of Equation D.5 as $\partial S/\partial g_i^{\star}$, treating g_i and $g_I^{\star}$ as independent variables. Evaluating $\frac{\partial S}{\partial g_i} = 0$ would give the complex conjugate of Equation D.21. Hence, $\partial S/\partial g_i$ gives no independent information not present in $\partial S/\partial g_i^{\star}$.