xtract: A Flexible Data Extraction Program For GMRT Visibility Database

Sanjay Bhatnagar
N.C.R.A., Pune

Jan, 1997

1 Introduction

The visibility data is an explicit function of time, baseline length, and frequency. Implicitly however, it is a function of many other parameters like the antenna co-ordinates, the RA, Dec of the fringe stopping center, local sidereal time (LST), compensating delays applied to the various antenna for fringe stopping etc. One would like to be able to extract any of these parameters from the database. In addition, as the visibility is a complex number, one may like to extract the data in (Re, Im) or (Amp, Phs) format.

Since the combination of parameters and the format in which they can be extracted is large, a program has been written to extract selected data/parameters from the GMRT native data format, referred to as the LTA-format. The contents and the format of the output data is programmable.

This document describes the design of this program. Section 2 is intended for "plain" users of the program while section 3 is targeted for more enterprising users who will find if useful to extend this program. Section 4 describes the application programmers interface to the underlying libraries used by this application. Section 5 describes the mechanism to extend the list of data/parameters which can be extracted.

Two of the keywords, the values of which require some explanation, are described below. For more details about the rest of the keywords, please refer to the online documentation of the program (which can be read by the command `explain` after starting the program).

2 User interface

The user interface[1] of this program is same as that provided by all other off-line programs which work on the LTA-database produced by the GMRT data acquisition system.

In the simplest form, when started, this program will list out all the parameters it depends upon. This will be presented to the user as a list of keywords and an interactive shell will be started. User can set/reset any of the keywords from this shell. The command `help` gives a list of commands available inside this shell and the command `explain` gives a more detailed explanation about the program and the various keywords that the application is sensitive to.

The output of `xtract` depends on the value of the keyword `fmt` (abbreviation for 'format').

[1] http://langur.ncra.tifr.res.in/home/sanjay/public_html/ui/ui.html

2.1 The `fmt` keyword

This keyword is used to supply a string which describes the contents as well as the format of the output data stream. It's syntax is a hybrid of the implicit loops of the `write` statement of `FORTRAN` (see the `FORTRAN` manual) and the formatting string used by the output functions of `C` (see documentation on `printf` in `C` manual).

Such a format string 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. Interferometer based numbers will automatically be a function of the antenna pair making the interferometer and one may want to extract data for a list of antennas, baselines, frequency channels and/or various time stamps.

We therefore need four variables for time, baseline, channels and antenna. All numbers extracted will then be a function of all or some of these variables. In our format, we name them "timestamp", "base", "chan" and "ant". Since all these can take a list of values, we call them operators in our terminology. The list of values for these operators is supplied as a list of comma (',') separated values for the keywords `timestamp`, `baselines`, `channels`, and `antenna` respectively.

The format string allows the user to write implicit loops and the loops are controlled by the list of values which each of the operator represent. These operators loop over the "body" of the loops for each value of the list. The body is a list of parameters separated by semicolon (';') enclosed in pair of curly braces ('{' and '}'). We refer to these parameters inside a "body" as the elements of the body. 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.

Out of these four operators, the user cannot have control on the "timestamp" operator. Therefore, the entire format string is implicitly the body of the time operator. This means that time in the output will vary at the slowest rate.

Various elements that the syntax recognizes are listed below. Some of these are independent of any operator while others need to be part of the body of one or more operators. The operators that these elements require are also listed with them.

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} (=$\mathrm{ua}_i - \mathrm{ua}_j$) 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.
delay,phs0	ant	The delay in μs applied to the antennas at the delay cards 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. `u` is equal to $U_1 - U_2$ where U_1 and U_2 are the co-ordinates of the two antennas making the baseline. `re,im` are real and imaginary parts of the visibility while `a,p` are the amplitude and phase. `cno` is an integer representing the frequency bin number. `ha,ist,lst` are Hour

angle, Indian Standard Time and Local Sidereal Time in hours respectively. `delay`, `phs0` are the antenna based delay in μs and phase applied for fringe rotation in degrees respectively.

Thus, to get the output such that each line is tagged with the HA value and has the data in Real,Imag format for all channels listed for the "chan" operator (via the keyword `channels`), and for all baselines listed for the "base" operator (via the keyword `baselines`), the `fmt` keyword should be set to

```
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 a `NEWLINE` here). The only other special element that this syntax currently allows is \\t (TAB).

If the U,V,W values for each baseline is also required to appear in each row, the `fmt` string would become

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

However note that the following setting for `fmt` has an 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. If this syntax is supplied, the program will generate an error message to this effect.

The elements can also be qualified with a `C`-styled `printf` format field. Hence, if the 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 any of the 'f','g','G','e', or 'E' type (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 2,3,4 will have the u,v,w values. Next 2*N will be real and imaginary values for the N channels listed in the "chan" operator.

  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 of a set of two rows of unequal lengths.

  The row one will have only HA and LST values.

  Row two 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 format will also generate a table of set of two rows of unequal lengths.

  Row number one will have HA, U, V, W values.

  Row number two will have real and imaginary values of the visibilities for all channels listed for the "chan" operator and for all values listed for the "base" operator.

2.2 The baselines keyword

This keyword provides mechanism to define selections on the baselines - either via the name of the baseline (explained below) or via a integer representing the baseline number. The selected baselines form the list of values for the base operator explained above.

The baseline names form a comma separated list of values, which could be either 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 full antenna name consist of three hyphen ('-') separated fields. First field is the antenna name, second is the side band name and the third is the polarization name. For e.g., a fully qualified name for C11, upper side band, 175 MHz. polarization channel would be:

```
C11-USB-175
```

A fully qualified baseline name for C11 and C12 antenna would be (upper side band, 175 MHz. polarization channel)

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

Antenna names can be regular expressions. Hence to choose all baselines with C11 upper side band and any polarization, one would use the baseline name as:

```
C11-USB-.+:.+
```

(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. For more detail about POSIX regular expressions, please read the document on regex. It helps to know this syntax since the `grep` (and certainly `egrep`) uses this syntax too.)

If only the first field of antenna name is provided, other fields are taken to be wild card characters.

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

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

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

2.2.1 Examples:

- All baselines with C11

  `baselines=C11`

- Self correlation of C11

  `baselines=C11:C11`

- Self correlation of C11 130 MHz polarization channel, any side band

  `baselines=AC11-.+-130`

- All USB baselines with C11

  `baselines=C11-USB-.+:.`

  or

  `baselines=C11-USB-.+`

- All USB baselines with C11 and baselines 10,15 and 18

  `baselines=C11-USB-.+,10,15,18`

2.3 Output filters

Most common use of `xtract` is to supply the data extracted from the LTA-file to some other program, typically a display package or some other program which processes the data further (for e.g., compute the antenna pointing errors). The output of `xtract` can be supplied to another program in two ways.

When the `out` keyword is left blank, the output of `xtract` is written to the standard output. Hence if `xtract` is started as

```
xtract | MyProg
```

the output of `xtract` will get piped to the standard input of the program named `MyProg`.

Second, and probably more convenient, method of piping data is to set the `out` keyword to '`|`' plus a string which should be a command which will read the output of `xtract`. For e.g., to pipe the data to a program called `MyProg`, set `out=|MyProg`. `xtract` will open a named pipe and write the data to it.

The output will be written in ASCII format, preceded by a simple header. The header will contain, apart from other fields, information about the number of rows and columns and the labels for each of the columns of the output table. This header always ends with a string `#End`, after which the data is written. A line begining 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.

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 can be accessed by setting `out=>QDP`. The out, in this case will be displayed as a stacked line plot by `QDP`.

3 Internal design

This program uses the Command-Line library (`cllib`[2]) for user interface and the off-line data analysis library for doing I/O to the visibility database written in the GMRT native data recording format.

To produce the output as specified by the format string, the program needs to interpret the format string. However, for efficiency reasons, the program interprets the format string and compiles it into a linked list. Once the format string is so compiled, the application "executes" this list for every selected time stamp.

The details about the compilation and execution of the format string are given below.

[2] http://langur.ncra.tifr.res.in/home/sanjay/public_html/uilib/uilib.html

3.1 Format compilation

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

First, all the loops in the format are exploded into a linked list (also called the symbol table). Each node of the symbol table represent an "element" and is 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 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 format string and then transfer the information about `Class` and `Type` to the actual symbol table.

Apart from the name of the element and the `C` styled format in which the value of the element would be written out as an ASCII string, the nodes of the symbol table also have information on the mechanism to get the numeric value associated with the element. This information is in the field `Type` which is initially transfered from the table of recognized symbols (see above). All elements need to be one of the following types:

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 carries the current values of the three operators (`ant`,`base`, and `chan`).

Before an element is put on the symbol table, check is made to ensure that all the required operators (as listed in the table above) are active. To generate this information about the required operators, elements are further categorized into one of the following classes:

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 the new link is filled with the `Name`, `Type` and `Class`. Since the loops (represented by the list of values of associated with various operators) are exploded, this is done for every value of the active operators and the current values of these operators are put in the `abc` array (passive operators have a value of -1).

The second step in the process of compilation is to fill in the information about the mechanism to get the numeric values of the elements on the list. By this time, if no error has occurred, it is assured that the syntax of the format was correct and all the elements in the format were recognized.

For filling in the above mentioned information, the `Type` of the element and, if required, the values in the `abc` array are used.

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

For `FTYPE` elements, 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 calculation of the value requires some data, the pointers to such data should be put in the `fargv` field and the total number of such pointer be filled 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 `CHARType` elements, nothing needs to be done. The name of such elements is the character that is to be copied on the output during execution.

3.2 Format execution

The process of "execution" of the compiled list of elements is rather simple. The application steps through the entire list of elements and checks the type of each element on the list.

If the type is `PTYPE`, the value pointed to by the `ptr` pointer is copied on the output using the format in the `Fmt` field of the element.

If the type is `FTYPE`, it makes a call to the function pointed to by the `func` field, with `Name`, `fargv`, and `fargc` as the arguments. The value returned by this function is then copied to the output using the format in the `Fmt` field.

If the type is CHARType, the first character of the Name field is copied on the output.

If the output in required in 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 the binary format. Such a routine is used when the extracted data is directly passed on to a plotting program.

Following is an example of a simple "Execute" routine:

```
/* $Id: xtract.tex,v 1.8 1998/06/04 11:21:32 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;
}
```

4 Programming with the xtract library

The process of compilation and execution of the format string is done by a stand-alone library. This section describes the usage of this library to compile and execute the format string to extract data from the GMRT visibility database.

To get all the definitions used by this, user must include `fmt.h` in the program. At link time, the applications must be linked to `libjump.a` in addition to all other GMRT Offline libraries (`liboff.a`, `libregex.a`, `libkum.a`).

4.1 Interpretation and compilation of the format string

The format string first needs to be interpreted. This is achieve by a call to:

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

First argument is a NULL terminated format string. Second argument is the fftmac[3] structure which holds the various mapping from sampler to baselines. Before it is used here, it must be filled using services

[3]http://langur.ncra.tifr.res.in/home/sanjay/public_html/gstruct/gstruct.html

provided by the GMRT Offline Library[4] (`getFFTMac method`). Third argument is a pointer to structure of the type `Parameters`. This structure has various parameters which the library uses later-on while executing the format string. Various fields of this structure are described later in this section (see section 4.3). This structure must be filled by the programmer before being used. Some of these parameters are defined by the user while others are to be extracted from the LTA-database.

Fourth argument is a pointer to a structure of type `SymbType`. This is the table of symbols used in the supplied format string. This should be initialized to NULL before being passed to this routine.

A return value of less than `EOF` (-1), indicates a syntax error in the format string.

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

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

First argument is the symbol table returned by a call to `interpret`. This now points to the head of a link list of `SymbType`. The last node of this list is NULL. Second argument is `fftmac` structure. Third argument is the table of antenna co-ordinates. This can be filled by services provided by the GMRT Offline Library[5] (`getFFTMac method`). Fourth argument is a pointer to the structure of type `Parameters`.

A return value of less than `EOF`(-1) indicates error in compilation of the format string. Otherwise, it returns the size of the compiled symbol table in units of the size of the structure `SymbType`.

4.2 Execution of the compiled format string

If the interpretation and compilation of the format string has been successfully completed, the compiled string can be executed via calls to user-supplied function with a signature

```
int Exec(FILE *fd, SymbType *Inst, float *Buf, int ProgSize).
```

`fd` is a pointer to a file already opened for writing to which the result of the execution of the format string is to be written. `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 version of this routine which will fill the data in `Buf`. `ProgSize` is the value returned by `Compile`. However this is generally not used.

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

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

- `Execute`

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

- `ExecuteDef`

  This function writes the output data to the buffer pointed by `Buf`. The size of the this buffer must be at least as long as the length of the symbol table (return value of `Compile`). This does not use the file descriptor.

- `ExecuteQDP`

  This function supplies output data to the `QDP` program via a pipe opened via the `libpipe.a` library. This uses the `Buf` pointer.

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

[4] http://langur.ncra.tifr.res.in/home/sanjay/public_html/ltaobj/ltaobj.html
[5] http://langur.ncra.tifr.res.in/home/sanjay/public_html/ltaobj/ltaobj.html

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 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*

  User selected list of the baseline, scan, antenna and channel numbers respectively. The channel numbers must be 0-relative and not the absolute channel index of the data base (which could start from any where in the band).

  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 these bit fields to list of integers representing the selected baselines. These functions are available from `liboff.a` and are described in Appendix A of GMRT Offline Library[6].

- *int NBase, NScans, NAnt, NChan*

  The lengths of `BList, SList, AList` and `CList`.

- *int dBNBase, dBNChan, dBNAnt*

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

- *int dBStartChan*

  The actual index of the first frequency channel in the data base.

- *int TimeOff, ParOff, DataOff*

  The offsets within the LTA-data buffer to locate time stamp, visibility parameters, and the visibility data itself. These are extracted from the database header.

[6] http://langur.ncra.tifr.res.in/home/sanjay/public_html/ltaobj/ltaobj.html

- *float sd,cd*

  The *sin* and *cos* of the declination of the pointing center of the telescope. This is used for the calculation of the (u, v, w) co-ordinates during execution.

- *float TUnits*

  The units of the time stamp. This is the multiplications factor by which the time stamp in the data must be multiplied to convert it into seconds of time. This is extracted from the database header.

- *float Lambda*

  The wavelength of the observing frequency in meters.

- *char *data*

  Pointer to the begining of the LTA-data buffer.

5 Adding new elements to the syntax

To add a new elements to the present syntax, one would need to define the `Name`, `Class`, `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 element. 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.