

GRIB APIs

Fortran 90 - C - Python interfaces

part 1

Dominique Lucas – Xavier Abellan Ecija
User Support

Content

- Introduction
- The GRIB API library
- Fortran 90 interface
- Migration from GRIBEX to GRIB API
- C interface
- Python interface
- References

Introduction

- **Do you really need to write a program to decode GRIB data?**
 - **Before converting/writing a (Fortran, C or Python) code, you should evaluate whether you could use some grib tools instead**
 - **Grib tools are optimized and fully supported**
 - **Grib tools are intended to cover the usual processing of GRIB data**
 - **Writing programs will remain necessary:**
 - **Encoding/decoding GRIB messages within a meteorological model or within some post-processing (e.g. MAGICS code).**
 - **Writing a program will usually be more efficient than trying to achieve the same with tools/scripts.**

The GRIB API library

- Written in ANSI C 99
- The only required package is jasper which enables the jpeg2000 packing/unpacking algorithm (GRIB-2). It is however possible to build GRIB API without jasper, by using the '--disable-jpeg' option at configuration (**not recommended**).
- The GRIB API has been ported and tested on all platforms available at ECMWF. We receive and welcome feedback from external sites and will include requested changes, fixes, etc ... where possible.

The GRIB API library

- The IFS is using the GRIB API. Metview, magics use the `grib_api`.
- We have started producing GRIB-2 data (on model levels) on 18 May 2011.
- The GRIB API tools used operationally.
- New data types (not only on Model levels) are also produced in GRIB-2 format, e.g. some MACC chemical data.
- Support:
 - Installation issues: software.support@ecmwf.int
 - Usage problems/questions: advisory@ecmwf.int

Usage at ECMWF

- The GRIB API library is available on all ECMWF platforms.
- One library for single and double precision. Within the library, everything is done in double precision. In Fortran 90, the GRIB API will return/use the precision of the data variables defined in your program.
- Three user interfaces supported: Fortran (90), C and Python
 - Fortran 90 interface: `use grib_api`
- (At ECMWF) Two environment variables `GRIB_API_INCLUDE` and `GRIB_API_LIB` are defined to ease the usage of GRIB API.
- On our platforms:
 - c2a/c2b: `xlf90 myprogram.f90 $GRIB_API_INCLUDE $GRIB_API_LIB`
 - ecgate: `gfortran myprogram.f90 $GRIB_API_INCLUDE $GRIB_API_LIB`
 - cca: `ftn myprogram.f90`

Usage at ECMWF – changing version

- On HPCs and ecgate:

```
module switch grib_api grib_api/<version>
```

- This year, we will use version **1.11.0**, the default version at ECMWF being version 1.10.0.
- See change history...

<https://software.ecmwf.int/wiki/display/GRIB/History+of+Changes>

General framework

- A (Fortran/C/Python) code will include the following steps:
 - Open one or more GRIB files (to read or write)
 - You **cannot** use the standard Fortran calls to open or close a GRIB file. You **have to** call **`grib_open_file/grib_close_file`**
 - Calls to load one or more GRIB messages into memory
 - These subroutines will return a unique **`grib identifier`** which you can use to manipulate the loaded GRIB messages
 - Calls to encode/decode the loaded GRIB messages
 - You can only encode/decode **loaded** GRIB messages
 - You should only encode/decode what you need (not the full message)
 - Calls to write one or more GRIB messages into a file (encoding only)
 - Release the loaded GRIB messages
 - Close the opened GRIB files

Particulars of the F90 GRIB API interface

- **All** routines have an optional argument for error handling:

```
subroutine grib_new_from_samples(igrib, samplename, status)  
  integer, intent(out)           :: igrib  
  character(len=*), intent(in)   :: samplename  
  integer, optional, intent(out) :: status
```

- **If status is not present and an error occurs, the program stops** and returns the error code to the shell.

Particulars of the F90 GRIB API interface

- Use *status* to handle errors yourself (e.g. necessary for MPI parallel codes)

```
call grib_new_from_samples(igrib, samplename, status)
```

```
if (status /= 0) then
```

```
    call grib_get_error_string(status,err_msg)
```

```
    print*, 'GRIB_API Error: ', trim(err_msg), ' (err=', status, ')'
```

```
    call mpi_finalize(ierr)
```

```
    stop
```

```
end if
```

Input arguments
Output arguments

- The exit codes and their meanings are available [here](https://software.ecmwf.int/wiki/display/GRIB/Error+codes) (or <https://software.ecmwf.int/wiki/display/GRIB/Error+codes>).

Loading/Releasing a GRIB message (1/2)

- It is absolutely necessary to load a message because the GRIB API can only encode/decode loaded GRIB messages

- 3 main subroutines to load a GRIB message

Input arguments
Output arguments

- `grib_new_from_file`(`ifile`, `igrib`)

Loads a GRIB message from a file already opened with `grib_open_file` (use `grib_close_file` to close this file)

- `grib_new_from_samples`(`igrib`, "GRIB1")

*Name of an existing
GRIB sample*

Loads a GRIB message from a sample. Used for encoding. See further ...

- `grib_new_from_index`(`indexid`, `igrib`)

Loads a GRIB message from an index. This index will first have to be built. See further ...

Loading/releasing a GRIB message (2/2)

- The above subroutines will return a **unique grib identifier** (*igrib*). You will manipulate the loaded GRIB message through this identifier.
- **You do not have access to the buffer containing the loaded GRIB message.** This buffer is **internal** to the GRIB API library.
- The buffer occupied by any GRIB message is kept in memory.
- Therefore, the routine **`grib_release(igrib)`** **should always** be used to free the buffer containing a loaded GRIB message.

Input arguments

Output arguments

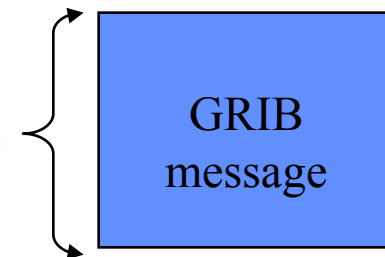
Example – Load from file

Input arguments
Output arguments

```
1 PROGRAM load_message
2   USE grib_api
3   implicit none
4
5   INTEGER                               :: rfile, igrib
6   CHARACTER(LEN=256), PARAMETER         :: input_file='input.grb'
7   CHARACTER(LEN=10), PARAMETER         :: open_mode='r'
8
9   !
10  ! Open GRIB data file for reading.
11  !
12  call grib_open_file(rfile, input_file, open_mode)
13
14  call grib_new_from_file(rfile, igrib)
15
16
17  call grib_release(igrib)
18  call grib_close_file (rfile)
19  END PROGRAM load_message
```

'r' to read, 'w' to write, 'a' to append (C naming convention)

Unique link to the buffer loaded in memory. Calls to grib_get/grib_set subroutines are necessary to access and decode/encode this message



Decoding a loaded GRIB message

- The idea is to decode as little as possible! You will never decode the whole **loaded GRIB message**.

(use 'grib_dump -D <grib_file>' to see how many keys there are!)

Input arguments

Output arguments

- One main subroutine to decode:

```
grib_get(igrib, keyname, keyvalues, status)  
integer, intent(in)           :: igrib  
character(len=*), intent(in)  :: keyname  
<type>, [dimension(:),] intent(out) :: keyvalues  
integer, optional, intent(out) :: status
```

Where <type> is integer or single/double real precision or string

Decoding a GRIB message – helper routines

- Get the size of an [array] key:

`grib_get_size(igrib, keyname, size, status)`

Input arguments
Output arguments

- Dump the content of a grib_message:

`grib_dump(igrib, status)`

- Check if a key is missing or not:

`grib_is_missing(igrib, keyname, missing, status)`

- Count messages in file:

`grib_count_in_file(ifile, count, status)`

Example – grib_get

! Load all the GRIB messages contained in file.grib1

call **grib_open_file**(ifile, 'file.grib1','r')

n=1

call **grib_new_from_file**(ifile,igrib(n), iret)

LOOP: do while (iret /= GRIB_END_OF_FILE)

 n=n+1; call **grib_new_from_file**(ifile,igrib(n), iret)

end do LOOP

! Decode/encode data from the loaded message

read*, indx

! Choose one loaded GRIB message to decode

call **grib_get**(igrib(indx) , “dataDate” , date)

call **grib_get**(igrib(indx), “typeOfLevel” , typeOfLevel)

call **grib_get**(igrib(indx), “level” , level)

call **grib_get_size**(igrib(indx), “values” , nb_values); allocate(values(nb_values))

call **grib_get**(igrib(indx), “values” , values)

print*, date, levelType, level, values(1), values(nb_values)

! Release

do i=1,n

 call **grib_release**(igrib(i))

end do

deallocate(values)

call **grib_close_file**(ifile)

Loop on all the messages in a file.

A new grib message is loaded from file. igrib(n) is the grib id to be used in subsequent calls

Values is declared as

real, dimension(:), allocatable:: values

GRIB API can do more ...

- The idea is to provide a set of high-level keys or subroutines to derive/compute extra information from a loaded GRIB message
- For instance:
 - keys (READ-ONLY) to return average, min, max of values, distinct latitudes or longitudes ...
 - Subroutines to compute the latitude, longitude and values:
`call grib_get_data(igrib,lats,lons,values,status)`
 - Subroutines to extract values closest to given geographical points:
`call grib_find_nearest(igrib, is_lsm, inlat, inlon, outlat, outlon, value, distance, index, status)`
 - Subroutine to extract values from a list of indexes: Input arguments
`call grib_get_element(igrib, key, index, value, status)` Output arguments

How to migrate from GRIBEX to GRIB API?

- All your new GRIB related developments should be made with the GRIB API library.
- For existing codes, the most difficult task is to find a correspondence between GRIBEX ksec0(*), ..., ksec4(*), psec2(*),...psec4(*), and GRIB API keys. See conversion tables under:

<https://software.ecmwf.int/wiki/display/GRIB/GRIBEX+keys>

- Try to use the “**recommended**” keys i.e. keys that are valid for both GRIB-1 and GRIB-2 (for instance “dataDate” instead of “YearOfCentury”, “month”, “day”). See edition independent keys under:

<http://www.ecmwf.int/publications/manuals/d/gribapi/keys/>

C API - Specifics

- `#include "grib_api.h"`
- module load `grib_api` to set `GRIB_API` paths
- Compilation on IBM platforms (HPC):
`xlC myprogram.c $GRIB_API_INCLUDE $GRIB_API_LIB -lm`
- Compilation on ecgate or other Linux based platforms:
`gcc myprogram.c $GRIB_API_INCLUDE $GRIB_API_LIB -lm`

Main differences:

- C-Interface is more **explicit** than FORTRAN, e.g. type specific functions such as `grib_get_double()`
- Some helper functions do not exist, e.g. `grib_open_file()`

C API - Error checking

- Built-in error checking for a call to a GRIB API function can be achieved with `GRIB_CHECK`:

```
GRIB_CHECK(grib_get_long(h,"Ni",&Ni),0);
```

- Or if error code is passed as function argument:

```
h = grib_handle_new_from_file(0,in,&err);
```

```
GRIB_CHECK(err,0);
```

- Alternatively, one can check the status:

```
err = grib_get_long(h,"Ni",&Ni);
```

```
if (err != 0) {
```

```
    printf("Error: %s \n unable to get long  
variable\n",grib_get_error_message(err));
```

```
    exit(1); }
```

C API - Loading/Releasing a GRIB message

- `grib_handle *h;`
 - structure giving access to parsed grib values.
- `grib_handle* grib_handle_new_from_file(grib_context *c, FILE *fp, int *error)`
 - Create handle from a file.
 - requires the input file to be opened with `fp=fopen(filename,"r")` and `fclose(fp)` to release the file pointer at the end of the program.
 - `grib_context *c` should usually be set to 0.
- `grib_handle * grib_handle_new_from_...`
 - `samples (grib_context *c, const char *res_name)` a message contained in a samples directory.
 - `message (grib_context *c, void *data, size_t length)` a user message in memory.
- `grib_handle_delete(h);`
 - Releases the handle.

C API – Decoding

- **int `grib_get_double`(grib_handle *h, const char *key, double *value)**
 - Gets a double value from a key, if several keys of the same name are present, the last one is returned. Similar function for `long` exists.
- **int `grib_get_string`(grib_handle *h, const char *key, char *message, size_t *length)**
 - Gets a string value from a key, if several keys of the same name are present, the last one is returned. String will be stored in pre-allocated memory message of size `length`. Similar function for `bytes` exists.
- **int `grib_get_double_array`(grib_handle *h, const char *key, double *values, size_t *length)**
 - Gets double array values from a key.
 - Use “int `grib_get_size`(grib_handle *h, const char *key, size_t *length)” to get the number of coded values (`length`) from a key (if several keys of the same name are present, the total sum is returned). Similar function for `long array` exists.

C API – Decoding

- `int grib_nearest_find(grib_nearest * nearest, grib_handle * h, double inlat, double inlon, unsigned long flags, double * outlats, double * outlons, double * values, double * distances, int * indexes, size_t * len)`
 - Finds the 4 nearest points and their distance of a given latitude longitude point.
 - `flags` should be set to 0, or can be set to
 - `GRIB_NEAREST_SAME_POINT`
 - `GRIB_NEAREST_SAME_GRID`
 - `outlats`, `outlons`, `values`, `distances` and `indexes` have to be allocated and `len` should be set to 4

Python interface – Specifics

- To import, use: `import gribapi`
- module load `grib_api`
- Low level, procedural
- Provides almost 1 to 1 mappings to the C API functions
- Uses the NumPy module to efficiently handle data values

Python interface - Loading/Releasing a GRIB message

gid = ***grib_new_from_file***(*file*, *headers_only=False*)

- Returns a handle to a GRIB message in a file
- Requires the input file to be a Python file object
- The use of the *headers_only* option is not recommended at the moment

gid = ***grib_new_from_samples***(*samplename*)

- Returns a handle to a message contained in the *samples* directory

gid = ***grib_new_from_message***(*message*)

- Returns a handle to a message in memory

grib_release(*gid*)

- Releases the handle

Python interface - Decoding

value = ***grib_get***(*gid*, *key*, *type=None*)

- Returns the value of the requested key in the message *gid* is pointing to in its native format. Alternatively, one could choose what format to return the value in (*int*, *str* or *float*) by using the *type* keyword.

values = ***grib_get_array***(*gid*, *key*, *type=None*)

- Returns the contents of an array key as a NumPy ndarray or Python array. *type* can only be *int* or *float*.

values = ***grib_get_values***(*gid*)

- Gets data values as 1D array

On error, a **GribInternalError** exception (which wraps errors coming from the C API) is thrown.

Python interface - Utilities

```
[outlat, outlon, value, distance, index] =  
grib_find_nearest(gid, inlat, inlon, is_lsm=False, npoints=1)
```

- Find the nearest point for a given lat/lon
- (Other possibility is npoints=4 which returns a list of the 4 nearest points)

```
iter_id = grib_iterator_new(gid,mode)
```

```
[lat,lon,value] = grib_iterator_next(iterid)
```

```
grib_iterator_delete(iter_id)
```

References

- GRIB-1, GRIB-2:

<http://www.wmo.int/pages/prog/www/WMOCodes.html>

- GRIB API:

<https://software.ecmwf.int/wiki/display/GRIB/GRIB+API/>

- GRIB API [Fortran](#), [C](#) or [Python](#) interfaces:

f90: https://software.ecmwf.int/wiki/display/GRIB/Fortran+package+grib_api

C: <https://software.ecmwf.int/wiki/display/GRIB/Module+Index>

Python: <https://software.ecmwf.int/wiki/display/GRIB/Python+package+gribapi>

- GRIB API examples:

<https://software.ecmwf.int/wiki/display/GRIB/Grib+API+examples>

- GRIBEX – GRIB API conversion

<https://software.ecmwf.int/wiki/display/GRIB/GRIBEX+keys>