

# ecCodes

## GRIB Fortran 90 - Python APIs Part 1

Dominique Lucas and Xavi Abellan

[Dominique.Lucas@ecmwf.int](mailto:Dominique.Lucas@ecmwf.int) [Xavier.Abellan@ecmwf.int](mailto:Xavier.Abellan@ecmwf.int)

# Content

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

# Introduction

- Do you really need to write a program to decode GRIB or BUFR data?
  - Before converting/writing a (Fortran, C or Python) code, you should evaluate whether you could use some tools included with ecCodes instead
  - 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.

# Usage at ECMWF

- The ecCodes library is available on all ECMWF platforms. Version 2.10.0 is loaded by default with 'module'.
- One single library for single and double precision. Within the library, everything is done in double precision. In Fortran 90, the ecCodes 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 eccodes'
  - Python interface: 'from eccodes import \*'
- (At ECMWF) Several environment variables, e.g. ECCODES\_INCLUDE or ECCODES\_LIB, are defined to ease the usage of ecCodes. For compatibility with the GRIB\_API, we've kept the GRIB\_API environmental variables. These point to the ecCodes directories.
- If you want to use the GRIB\_API, you should unload ecCodes:

```
$ module swap eccodes grib_api
```

# Usage at ECMWF

- Compilation of ecCodes at ECMWF:

```
ecgate$ gfortran myprogram.f90 $ECCODES_INCLUDE $ECCODES_LIB
```

```
cca/ccb$ ftn myprogram.f90
```

- Changing version of ecCodes at ECMWF:

```
$ module switch eccodes/<version>
```

- See change history ...

<https://confluence.ecmwf.int/display/ECC/Latest+news>

# Names of ecCodes routines to handle GRIB data

- All routines in ecCodes start with 'codes\_', e.g.  
call codes\_open\_file(infile,'data.grib')
- Most routines apply both to GRIB and BUFR data, e.g.  
call codes\_get(igrib,'date')  
call codes\_get(ibufr,'typicalDate')  
call codes\_any\_new\_from\_file(ifile, iany)  
call codes\_new\_from\_file(ifile, product\_kind, kind)
- Some routines are different for GRIB and BUFR, e.g.  
call codes\_grib\_new\_from\_file(ifile, igrib)  
call codes\_bufr\_new\_from\_file(ifile, ibufr)
- The GRIB API routine names starting with 'grib\_' also exist in ecCodes, e.g.  
call grib\_open\_file(infile,'data.grib')

# 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.
  - 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
- This framework is also valid for BUFR data.

# Particulars of the F90 ecCodes interface

- All routines have an optional argument for error handling:

subroutine codes\_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 ecCodes interface

- Use status to handle error yourself, e.g. necessary for MPI parallel codes:

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

Input arguments

Output arguments

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

```
    call codes_get_error_string(status, err_msg)
```

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

```
    call mpi_finalize(ierr)
```

```
    stop
```

```
end if
```

- The exit codes and their meanings are available under:

[http://download.ecmwf.int/test-data/eccodes/html/group\\_errors.html](http://download.ecmwf.int/test-data/eccodes/html/group_errors.html)

# Loading/Releasing a GRIB message (1/2)

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

- 3 main subroutines to load a GRIB message

- call `codes_grib_new_from_file(ifile, igrib)`

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

- call `codes_grib_new_from_samples(igrib, "GRIB1")`

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

- call `codes_new_from_index(indexid, igrib)`

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

Input arguments

Output arguments

*Name of an existing  
GRIB sample*

# Loading/releasing a GRIB message (2/2)

- The above 3 'loading' 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 ecCodes library.
- The buffer occupied by any GRIB message is kept in memory.
- Therefore, the routine **codes\_release**(*igrib*) **should always** be used to free the buffer containing a loaded GRIB message when it is no longer needed.

Input arguments

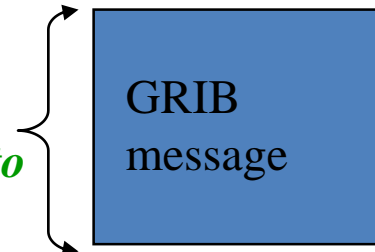
Output arguments

# Example – Load from file

```
1 PROGRAM load_message
2   USE eccodes
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 codes_open_file(rfile, input_file, open_mode)
13
14  call codes_grib_new_from_file(rfile, igrib)
15
16
17  call codes_release(igrib)
18  call codes_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 ecCodes keys there are!)

- One main subroutine to decode:

```
codes_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

Input arguments

Output arguments

# Decoding a GRIB message – helper routines

Input arguments  
Output arguments

- Get the size of an [array] key:

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

- Dump the content of a grib\_message:

```
codes_dump(igrib, status)
```

- Check if a key is defined (exists) or not:

```
codes_is_defined(igrib, keyname, exists, status)
```

- Check if the value for a key is missing or not:

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

- Count messages in file:

```
codes_count_in_file(ifile, count, status)
```

# Example – codes\_get

! Load all the GRIB messages contained in file.grib1

call codes\_open\_file(ifile, 'file.grib1', 'r')

n=1

call codes\_grib\_new\_from\_file(ifile, igrib(n), iret)

LOOP: do while (iret /= CODES\_END\_OF\_FILE)

    n=n+1; call codes\_grib\_new\_from\_file(ifile, igrib(n), iret)

end do LOOP

*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*

! Decode/encode data from the loaded message

read\*, indx

! Choose one loaded GRIB message to decode

call codes\_get( igrib(indx) , “dataDate”, date)

call codes\_get(igrib(indx), “typeOfLevel”, typeOfLevel)

call codes\_get(igrib(indx), “level”, level)

call codes\_get\_size(igrib(indx), “values”, nb\_values); allocate(values(nb\_values))

call codes\_get(igrib(indx), “values”, values)

print\*, date, levelType, level, values(1), values(nb\_values)

*Values is declared as*

*real, dimension(:), allocatable:: values*

! Release

do i=1,n

    call codes\_release(igrib(i))

end do

deallocate(values)

call codes\_close\_file(ifile)

# 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
```

*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*

```
! 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)
```

*Values is declared as*

*real, dimension(:), allocatable:: values*

```
! Release
```

```
do i=1,n
```

```
    call grib_release(igrib(i))
```

```
end do
```

```
deallocate(values)
```

```
call grib_close_file(ifile)
```



# ecCodes 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, offsets, identifier (GRIB, BUFR) ...

- Subroutines to compute the latitude, longitude and values:

```
call codes_get_data(igrib,lats,lons,values,status)
```

- Subroutines to extract values closest to given geographical points:

```
call codes_grib_find_nearest(igrib, is_lsm, inlat, inlon, outlat, outlon, value, distance, index, status)
```

- Subroutine to extract values from a list of indexes:

```
call codes_get_element(igrib, key, index, value, status)
```

Input arguments

Output arguments

# How to migrate from GRIBEX to ecCodes?

- GRIBEX is no longer included in the default user environment at ECMWF. It was removed 2 years ago. It is no longer supported.
- If you still have GRIBEX codes, the essential task is to find the corresponding ecCodes key names to the elements in the GRIBEX ksec0(\*), ..., ksec4(\*), psec2(\*),...psec4(\*) header sections. See conversion tables under:

<https://confluence.ecmwf.int/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://apps.ecmwf.int/codes/grib/format/edition-independent/>

For GRIB data, the main difference between the GRIB API and ecCodes packages is ...



See <https://confluence.ecmwf.int/display/ECC/GRIB-API+migration>

# How to migrate from GRIB API to ecCodes?

- There should be very little to do to your GRIB API codes.
- You could replace the calls to `grib_` routines by calls to `codes_` routines.
- At ECMWF, ecCodes has replaced the GRIB API as default GRIB decoder/encoder since May 2017.
- New GRIB API developments are only done in ecCodes.
- GRIB API support has been discontinued at the end of 2018.
- <https://confluence.ecmwf.int/display/ECC/GRIB-API+migration>

# Python API - Specifics

- To import, use:

```
import gribapi
import eccodes
```

```
from gribapi import *
from eccodes import *
```

- @ ECMWF: module load eccodes.
  - If grib\_api is loaded, then you should module swap grib\_api eccodes
- Low level, procedural
- Provides almost 1 to 1 mappings to the C API functions
- Uses the NumPy module to efficiently handle data values
- From version 2.10.0, it supports Python 3 (BETA)

# Python API – Loading/Releasing a GRIB message

gid = *codes\_grib\_new\_from\_file* (file, headers\_only=False)

*codes\_any\_new\_from\_file*

*codes\_new\_from\_file* (file, **product\_kind**)

- CODES\_PRODUCT\_GRIB
- CODES\_PRODUCT\_BUFR
- CODES\_PRODUCT\_ANY

*grib\_new\_from\_file*

Returns a handle to a GRIB message in a file.  
Requires the input file to be a Python file object open in binary mode.

gid = *codes\_new\_from\_samples* (samplename)

Returns a handle to a message contained in the samples directory

*grib\_new\_from\_samples*

gid = *codes\_new\_from\_message* (message)

Returns a handle to a message in memory

*grib\_new\_from\_message*

*codes\_release* (gid)

Releases the handle

*grib\_release*

# Python API – Decoding

```
value = codes_get (gid, key, ktype=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.

***grib\_get***

```
values = codes_get_array (gid, key, ktype=None)
```

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

***grib\_get\_array***

```
values = codes_get_values (gid)
```

Gets data values as 1D array

***grib\_get\_values***

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

***GribInternalError***

# Python API – Utilities

[outlat, outlon, value, distance, index] =

***codes\_grib\_find\_nearest*** (gid, inlat, inlon, is\_lsm=False, npoints=1)

Find the nearest point for a given lat/lon

With npoints=4 it returns a list of the 4 nearest points

***codes\_find\_nearest***

iter\_id = ***codes\_grib\_iterator\_new*** (gid,mode)

***grib\_iterator\_new***

[lat,lon,value] = ***codes\_grib\_iterator\_next*** (iterid)

***grib\_iterator\_next***

***codes\_grib\_iterator\_delete*** (iter\_id)

***grib\_iterator\_delete***



# References

- GRIB-1, GRIB-2:

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

- ecCodes:

<https://confluence.ecmwf.int/display/ECC/ecCodes+Home>

- ecCodes [Fortran](#), [C](#) or [Python](#) interfaces to GRIB data (currently on the GRIB API wiki):

<https://confluence.ecmwf.int/display/ECC/ecCodes+API+Reference>

- Examples:

<https://confluence.ecmwf.int/display/ECC/GRIB+examples>

- GRIBEX to ecCodes conversion:

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

- GRIB API to ecCodes conversion:

<https://confluence.ecmwf.int/display/ECC/GRIB-API+migration>