Specifications

for the

IMOS USER CODE LIBRARY

Version 1.0

Prepared by *Laurent Besnard* and *Guillaume Galibert*

IMOS - eMII

15/04/2013

IMOS

Integrated **Marine**
Observing System

Table of Contents

Revision History

Name	Date	Reason For Changes	Version

1. Introduction

1.1 Context

IMOS (Integrated Marine Observing System) is designed to be a fully integrated national array of observing equipments monitoring the open oceans and coastal marine environment around Australia, covering physical, chemical and biological variables. All IMOS data is freely and openly available through the IMOS Ocean Portal for the benefit of Australian marine and climate science as a whole.

Most of the IMOS data is stored as NetCDF files (cf http://www.unidata.ucar.edu for more information regarding NetCDF).

Some IMOS data users are not familiar with the netCDF format. A wide range of tools is already available online to help NetCDF users, but there is also a need for users who don't know where to start to include NetCDF files in their workflow.

The purpose of the "IMOS user code library" is to provide a ready to go code solution to incorporate data from NetCDF files in their working environment, starting with a NetCDF parser.

In its first release, users should be able:

- from a single function access an IMOS netCDF file either locally or from an OPeNDAP URL and retrieve all the data and metadata contained in it into a variable structure that is easy to manipulate in the given programming environment.

1.2 Purpose of document

This document aims to provide specifications for the "IMOS user code library" and its first component the NetCDF parser. It will also provide whoever would like to contribute in writing or improving the "IMOS user code library" with pre-requisite information and coding best practices. This document lays out the strategy for use in any programming language.

1.3 Future directions

The first release of this document comes with a NetCDF parser, capable of parsing a NetCDF file either locally of from an OPeNDAP URL. In future releases, more features could be added, such as :

- Sub-setting a NetCDF file
- Outputting the obtained data to a CSV file
- Creating some basic plots (time-series, profile, TS)
- RAMADDA search implementation of a specific data set
- iPython Notebook : a python console directly available as a web interface. For example IMOS staff could run directly some examples from this application during Data User Workshops http://ipython.org/ipython-doc/dev/interactive/htmlnotebook.html

1.4 Resources

A number of documents written by IMOS are useful resources for anyone who would like to contribute to this project:

Facility Manuals and Handbooks (http://imos.org.au/facility_manuals.html):

- IMOS NetCDF Manual
- IMOS Filename Convention

The followings links are access to different IMOS products, which could be useful in terms of code testing:

- The IMOS toolbox used to convert raw files from instruments to IMOS compliant NetCDF files:
  http://code.google.com/p/imos-toolbox/
- IMOS Portal:
  http://imos.aodn.org.au/webportal/
- IMOS THREDDS catalog
  http://thredds.aodn.org.au/thredds/catalog/IMOS/

2. Library Requirements and Pre-requisites

2.1 Contributing to the code library

A public git repository is available at:
https://GitHub.com/aodn/imos_user_code_library/

Having a GitHub repository will help developers to update the codes and keep track of the changes (such as with svn). It is also possible to contribute to this project by forking the repository. Users can download a full repository easily as a zip file without being logged in to GitHub.

For more information about how to interact with GitHub, please take a look at this help page
https://help.GitHub.com/articles/fork-a-repo

Any person who would like to contribute to the main project will have to contact IMOS / eMII in order to become a developer of this project. Please send an email to info@emii.org.au and mention in your email you would like to contribute to the "IMOS user code library" project.

The developer will be granted with relevant permissions to work within this repository using a GitHub client.

2.2 GitHub Repository Structure

The most common tools/codes which would be the same for all the different data-types will be placed in:
https://github.com/aodn/imos_user_code_library/[programing-language-name]/commons

and then ordered by areas such as :
https://github.com/aodn/imos_user_code_library/[programing-language-name]/commons/NetCDF/
https://github.com/aodn/imos_user_code_library/[programing-language-name]/commons/plotting/
https://github.com/aodn/imos_user_code_library/[programing-language-name]/commons/ramadda/
https://github.com/aodn/imos_user_code_library/[programing-language-name]/commons/examples/
…

and where [programing-language-name] can be :

- MATLAB_[VERSION]
- R_[VERSION]
- PYTHON_[VERSION]

So that we could have a directory named MATLAB_R2008 for example.

A 'Where_to_Start.txt' text file, which has general information and hints for new users, and suggests which functions to start with could be added in the top directory (before the [programing-language-name] folder). This is very necessary to be equivalent for all languages.

For more specific tools, a developer might have to create some codes only applicable to a unique dataset / facility / sub-facility. In this case, this set of new scripts has to be put in a facility/sub-

facility folder. It is essential to keep the same folder structure as the one which already exists on the THREDDS catalog and on the portal menu, for consistency.

For example, a user wants to plot a NetCDF file from a Bio-Optical dataset which needs specific plotting tools (The Bio-Optical database is a sub-facility of SRS). The script folder structure on GitHub should be:
https://github.com/aodn/imos_user_code_library/[programing-language-name]/SRS/BioOptical/
in order to host the specific scripts.

Once the code's location is set up, it is important to follow the same directory structure and filenames across programing languages.

2.3 Folder Documentation

The content of each of the GitHub sub-directories requires a **FUNCTION_SUMMARY.txt** file which lists all the available functions and gives a quick abstract about them as well as a tutorial. The text below is a FUNCTION_SUMMARY.txt file example:

> FUNCTION_SUMMARY.txt
>
> THIS FILE LISTS ALL THE FUNCTIONS AVAILABLE IN THE CURRENT FOLDER, GIVES A SHORT DESCRIPTION AND A HOWTO
>
> Contents
>
> 1 Function 1
> 1.1 Installation
> 1.2 Dependencies
> 1.3 Description
> 1.4 Use example
> 1.5 Known Issues
> 1.6 Contact
> etc...

2.4 Files Header

As well as a FUNCTION_SUMMARY file, each single function, or file needs to be properly documented. The header of each new script file should follow the following templates or equivalent for other languages:

MATLAB file header :

```
%FUNCTION_NAME - One line description of what the function or script performs (H1
%line)
%Optional file header info (to give more details about the function than in the H1 line)
%Optional file header info (to give more details about the function than in the H1 line)
%Optional file header info (to give more details about the function than in the H1 line)
%
% Syntax:  [output1,output2] = function_name(input1,input2,input3)
%
% Inputs:
```

```
%    input1 - Description
%    input2 - Description
%    input3 - Description
%
% Outputs:
%    output1 - Description
%    output2 - Description
%
% Example:
%    Line 1 of example
%    Line 2 of example
%    Line 3 of example
%
% Other m-files/library required: none
% Subfunctions: none
% MAT-files required: none
%
% See also: OTHER_FUNCTION_NAME1,  OTHER_FUNCTION_NAME2
%
% Author:
% Institute:
% email address:
% Website:
% December 2012; Last revision: 30-Nov-2012
%
% Copyright 2013 IMOS
% The script is distributed under the terms of the GNU General Public License
```

PYTHON File header:

```
#!/bin/env python
# -*- coding: utf-8 -*-
#
# Author:
# Institute:
# email address:
# Website:
# December 2012; Last revision: 30-Nov-2012
#
# Copyright 2013 IMOS
# The script is distributed under the terms of the GNU General Public License
```

Python Function header (directly after **def** statement):

```
#FUNCTION_NAME - One line description of what the function or script performs (H1
#line)
#Optional file header info (to give more details about the function than in the H1 line)
#Optional file header info (to give more details about the function than in the H1 line)
#Optional file header info (to give more details about the function than in the H1 line)
#
# Syntax:  [output1,output2] = function_name(input1,input2,input3)
#
```

```
# Inputs:
#    input1 - Description
#    input2 - Description
#    input3 - Description
#
# Outputs:
#    output1 - Description
#    output2 - Description
#
# Example:
#    Line 1 of example
#    Line 2 of example
#    Line 3 of example
#
# Other python-files/library required: none
# Subfunctions: none
#
# See also: OTHER_FUNCTION_NAME1,  OTHER_FUNCTION_NAME2
```

2.5 Coding best practices

Among many important properties, this "IMOS user code library" should be an example of well-written code. This is especially important considering many users of the library will be new to programming. It will also make it easier for users to contribute to the code.

As a general rule, the code must be well documented, clear, concise and robust. It is wiser to start with simple short functions before thinking about more complex features.

Some of the coding highlights are:

- Choice of external APIs / libraries
  As a general rule, it is recommended, when available, to use the most commonly used, active and up to date external library or API if there is a need to include any in the "IMOS user code library".

  For example, if there is no native NetCDF support in your targeted programming language, it is recommended when possible to use UNIDATA's API when dealing with NetCDF file format.
  One of the recommended toolbox in order to have a non-native integration of OPeNDAP capabilities within MATALB is the ***NCTOOLBOX*** (http://code.google.com/p/nctoolbox/). The installation is easy and doesn't need any compilation. With Python the ***NetCDF4*** module can be used.

  In some cases, eMII might have to consider support for the use of distinct popular libraries. These external libraries/APIs don't have to be included in the GitHub repository. But it is necessary to write all the required steps to install an external API onto a new machine. Such information will be made available in the main README.txt file.

- Include comments to describe your code and some major sections of it. Function and variable names can be more informative than comments when clearly chosen
- Closing an opened file is important!
- Handle exceptions/errors using 'try-catch' statements

- Use a 'config' text file when relevant instead of letting the user modify critical variables inside the code. (example access to FTP server when login pwd … need to be provided)
- Check that the number and type of input arguments is correct
- If a developer wants to write an already existing function into a new programing language, the same function name should be used.

The following are some links to best programing practices for PYTHON and MATLAB developers.
http://www.python.org/dev/peps/pep-0008/
http://www.mathworks.com.au/matlabcentral/fileexchange/2529-matlab-programming-style-guidelines

3. NetCDF parser

The NetCDF parser function, named **ncParse**, is the core of the "IMOS user code library". This function parses a NetCDF file, either from a local address or an OPeNDAP URL, and harvests its entire content into the workspace of the programing language used.
It is important to consider both scenarios depending on the external library used to handle OPeNDAP and/or local files. For example, if coding with MATLAB, the ***nctoolbox*** can be used for both remote and local NetCDF files. With Python the ***NetCDF4*** module can be used, as well for both remote and local NetCDF file. In addition, because of the nature of these two libraries, the call to the function and so the code will be exactly the same whether a file is local or on an OPeNDAP server.
Both the metadata and data can be extracted. The output of the NetCDF parser can be easily used as an input to more specific functions.

Call : output = ncParse (inputFileName, ['*parserOption'* , parserOption] , ['varList' , varList])

The different inputs are described in the sections below.

The output will be a data structure that is easy to read and manipulate in the relevant programming environment. See the output format section 3.4.

The following sections will describe the NetCDF parser features.

3.1 Parse a whole file

Description:
The **ncParse** function is used to retrieve all metadata (global attributes and variables attributes) and data from the file.

Input:
- A string of the NetCDF file 'path/address' or opendap URL as the first argument
- and/or the optional argument '*parserOption'* = 'all' (' '*parserOption'* = 'all' is the default value. This is equivalent to not having this optional argument)

For example. :
ncParse('/path/to/netcdfFile.nc')
ncParse('/path/to/netcdfFile.nc' , '*parserOption'* , 'all')
Both calls are equivalent

Output:
Please refer to the Output format section 3.4. for full dataset structure documentation.

Details on algorithms:

Open/Close a NetCDF file
If the file is not accessible, the function stops and returns an error message "ncParse: File <NetCDF file> not found". The function needs to open the NetCDF file, and to close it afterwards even if the function has an error statement while running (try-catch statements to handle exceptions should be used).

Retrieve global attributes of a NetCDF file
Each global attribute and its respective value is harvested.

MATLAB:
The output is stored as a structure called 'dataset.metadata' and each global attribute becomes a field of this structure. For example if we have a global attribute called 'abstract' with the value 'this is the abstract' , the information is therefor stored as dataset.metadata.abstract = 'this is the abstract'

PYTHON:
In Python, a data-type similar to what MATLAB can offer with its structure data type is a dictionary, which could be used for the output of the NetCDF parser.

Retrieving attributes, values and quality control flags of variables in a NetCDF file.
In an IMOS NetCDF files, each main variable, such as TEMP (for temperature), should be linked in the NetCDF file to an ancillary variable of quality control flag values, for example TEMP_QUALITY_CONTROL.

In this NetCDF parser we have proposed to write, we have chosen to put the flag values straight into the main variable 'structure' and not in another variable, like in the NetCDF file. Those flag values will be stored in :**dataset.variables.<variable_shortname>.flags**

It is therefor essential to only look for the variables which don't have the string '_quality_control' (check for both upper and lower case scenarios) in their name. We only want to list the non quality control variables in the structure output.

For example, if there is a variable called 'TEMP' in the NetCDF file, one of the attribute of this variable should be 'ancillary_variables'. This attribute value is the name of the quality control variable which 'TEMP' is linked to. In this case it should be 'TEMP_quality_control'. If the the attribute 'ancillary_variables' is missing or empty, the developer needs to check if the variable 'TEMP_quality_control' at least exists, since the 'ancillary_variables' attribute is not mandatory. The QC values will be stored as shown below with the main 'TEMP' variable.

The dimensions of each variable are also retrieved in a separate branch of the structure. The dimensions are written in :
dataset.variables.<variable_shortname>.dimensions
which give the shortnames of the dimensions.
The output is stored as a structure called
*'dataset.variables.<variable_shortname>'*for the variables
and '*dataset.dimensions.<dimension_shortname>'* for the variable dimensions.

For example, assuming there is a variable called TEMP in a NetCDF file (The following data structure is specific to MATLAB but can be applicable to any other programming language) :

dataset.variables.TEMP=
dimensions: {'TIME'}
data: [16699x1 double]
data_CORR: [16699x1 double]
standard_name: 'sea_water_temperature'
long_name: 'sea_water_temperature'
units: 'Celsius'
FillValue: 999999
valid_min: 0

valid_max: 50
ancillary_variables: 'TEMP_quality_control'
quality_control_set: 1
flags: [16699x1 int8]
flag_meanings: [10x1 int8]
flags_values: [10x1 int8]
flag_quality_control_conventions

dataset.dimensions.TIME=
data: [16699x1 double]
standard_name: 'time'
long_name: 'time'
units: 'days since 1950-01-01 00:00:00 UTC'
axis: 'T'
valid_min: 0
valid_max: 90000
FillValue_: 999999
calendar: 'gregorian'
comment: 'timeOffsetPP: TIME dimension and time_coverage_start/end global attributes have been applied the following offset : +0 hours.'
quality_control_set: 1
quality_control_conventions
flags: [16699x1 int8]
flag_meanings: [10x1 int8]
flags_values: [10x1 int8]

As well as harvesting the values, it is important to convert some of them so they can be directly used:

- For each variable, replace the empty values of each variable with the current programming language NaN (Not a Number) value by reading the '_FillValue' attribute of each variable

- Add the offset and scale factor by reading the 'add_offset' and 'scale_factor' variable attributes when they exist according to this formula:
  `varData = varDataScaled*scale_factor+add_offset;`

- For a TIME variable, it is necessary to convert the TIME values stored in the NetCDF file into the current programming language time reference and unit. This is done by reading the 'units' attribute from the TIME variable. The 'units' value can vary amongst datasets. It is possible to have 'days since …' 'seconds since …' or 'hours since ...' . The rest of the string is always written in the same way 'YYYY-MM-DD hh:mm:ss UTC' (where DD=int(Day); MM=int(Month); YYYY=int(Year); hh=hour; mm=minute; ss=second)
  example : "days since 1950-01-01 00:00:00 UTC"

3.2 Parse Metadata only

Description:
The **ncParse** function can be used to retrieve only metadata (global attributes and variables attributes) from a NetCDF file and no data

Input:
- A string of the NetCDF file path/address as the first argument,
- and the optional argument '*parserOption'* = 'metadata'
  in order to retrieve only the metadata.(cf. section 3.1 'parse a whole file' section')

ex. : ncParse('/path/to/netcdfFile.nc','*parserOption'* = 'metadata')

Output:
Documented in Section 3.4, for global and variable attributes only.

Details on algorithms:
For more details please read **Retrieve global attributes of a NetCDF file** in the Section 3.1 above

3.3 Parse only a specified set of variables

Description:
The **ncParse** function can be used to retrieve only a specified set of variables . This is an optional feature.

Input:
- A string of the NetCDF file 'path/address' or URL
- the optional argument *'varList'* listing the variables to be returned.

For example:
- ncParse('/path/to/netcdfFile.nc' , 'varList' , {'PSAL' , 'TEMP'})
  will only grab data and metadata for both PSAL and TEMP + dimensions data and metadata of PSAL and TEMP.

which is the same as:

ncParse('/path/to/netcdfFile.nc' , 'parserOption' , 'all', 'varList' , {'PSAL' , 'TEMP'})

- ncParse('/path/to/netcdfFile.nc' , 'parserOption' , 'metadata', 'varList' , { 'TEMP'})
  will parse all metadata only for TEMP + dimensions metadata of TEMP.

Details on algorithms:
If a variable name does not exist but is written in the list '*varList*', the function will return a warning such as:
'ncParse: WARNING variable <variable> does not exist in netcdf file <NETCDF file>'
and will act as if the variable was not in the list.
For more details, please read section 3.1.Parse a whole file above.

3.4 Output format

This section summaries the output format of the netCDF parser. Although the parser output is language-specific, this section provides a detailed summary and clear guidelines.

dataset.metadata.<attributeName>='attributeValue'
- netcdf_filename (this is the filename of the original NetCDF file)

- plus other attributes

dataset.variables.<variable_shortname>.

- dimensions = [<dimension_shortname1> <dimension_shortname2> ...]
- data
- standard_name
- long_name
- units
- _FillValue
- valid_min
- valid_max
- ancillary_variables
- quality_control_set
- flags
- flag_meanings
- flag_values
- etc...
-

dataset.dimensions.<dimension_shortname>.

- data
- standard_name
- long_name
- units
- axis
- valid_min
- valid_max
- calendar
- comment
- quality_control_set
- quality_control_conventions
- flags
- flag_meanings
- flag_values
- etc...

This variable structure might be different depending on the programing language (ex Matlab vs Python <=> structures vs dictionaries), but the delivery of data to the programming language is the same.

The ncParse should return its output in a structure that makes it easy to:

- get the value of any single global or variable attribute;
- get all global attributes or all attributes of a variable in a list or array structure;
- get the values of any variable in a numerical array type that allows computations and plotting;