Copernicus Climate Change Service Global Land and Marine Observations Database

Common Data Model Mapper

User manual

**Issued by:** NUIM / Peter Thorne

**Date:** 31/12/2018

**Ref:**C3S\_D311a\_Lot2.3.4.4-2018\_201812\_ Marine\_code\_v1

**Official reference number service contract:** 2017/C3S\_311a\_Lot2\_NUIM/SC1

Contributors

natural environment research council, National oceanography centre (NERC noc)

1. David I. Berry

2. Elizabeth C. Kent

natIONAL UNIVERSITY OF IRELAND MAYNOOTH

1. Peter Thorne

2. Corinne Voces

National oceanic and atmospheric administration’s national centers for environmental information

1. Eric Freeman

# Table of Contents

[1. Table of Contents 3](#_Toc11310799)

[2. Introduction 5](#_Toc11310800)

[3. Tool overview 6](#_Toc11310801)

[4. Use, input, output and syntax 7](#_Toc11310802)

[4.1 Input and output 7](#_Toc11310803)

[4.2 Syntax 8](#_Toc11310804)

[4.3 Combining data from different data models 9](#_Toc11310805)

[5. How to register a new data model mapping 10](#_Toc11310806)

[6. Data model mappings 12](#_Toc11310807)

[6.1 CDM tables mapping files and descriptors 12](#_Toc11310808)

[6.2 Mapping sequence 13](#_Toc11310809)

[6.3 Defining mapping functions 13](#_Toc11310810)

[6.4 Direct code table keys mapping 15](#_Toc11310811)

[7. CDM tables definition 16](#_Toc11310812)

[7.1 Updating tables from GLAMOD Common Data Model repository 16](#_Toc11310813)

[8. Replacing mappings 16](#_Toc11310814)

Executive Summary

The C3S 311a Lot 2 (Global Land and Marine Observations Database) service is concerned with the provision of globally available land and marine surface meteorological records. The service includes inventorying of, and brokering access to, data sources, their harmonization (via conversion to a Common Data Model (CDM), merging, and quality assurance) and their provision via the Copernicus Climate Change Service (C3S) Climate Data Store (CDS).

The version history is given below:

|  |  |  |
| --- | --- | --- |
| **Version** | **Release Date** | **Release notes** |
| 1.0 |  |  |
| 2.0 |  |  |

# Introduction

The Copernicus Climate Change Service (C3S) Global Land and Marine Observations Database service provides brokered access to global historical holdings of surface meteorological observations. It builds upon existing national, regional and global efforts to create an augmented set of quality assured holdings that can be used to create datasets, products and services.

**This document will contain relevant detailed information on the software and configuration used to produce the marine data holdings for service providers to be able to access, use, modify and/or update where necessary. The current version of this document is focused on the data flow leading to the December 2018 beta release (header and observations tables) and the scripts and software tools derived from it.**

The document is ordered as follows:

* Section 2 summarizes the marine code requirements.
* Section 3 describes the marine main data flows.
* The Annex to this document provides further insight into scripts and tools.

# Tool overview

*cdm* is a python3 tool designed to map observed variables and its associated metadata from a data model or models combination (*imodel*) to the C3S CDS Common Data Model (*CDM*) where:

* **Input data** is the *imodel* data elements in a unique pandas DataFrame with its attributes (data type, native precision-number of decimal places…) available in a python dictionary.
* **Output** is a set of ascii field delimited files each with one of the *CDM* tables for which a mapping has been defined. Optionally, the intermediate mapped python object can also be accessed.
* The full transformation from *imodel* to the *CDM* is ***internally*** defined in the *imodel* subdirectory of the mappings’ library by:
  + A set of *cdm\_table*.json files, defining the mappings to each *CDM* table from the *imodel.*
  + A *imodel*.py file with the full set of transforming functions to *CDM* elements from the *imodel*.
  + A set of json files with the code table mappings between the *imodel* and the *CDM*
* With the above settings, this tool supports mapping to *CDM* elements by:
  + Direct mapping from an *imodel* element
  + Direct assignment from the *imodel* element’s attributes
  + Assignment of a *default* value
  + Transformation: *imodel* element’s transformation including any combination of the following:
    - element combination
    - simple parametrization (input keyword arguments)
    - *imodel* element’s attributes
    - *cdm* element to *cdm* element: currently not supported.
  + *imodel* or CDM coded values:
    - From *imodel* key to CDM key (mmm… see nested dictionaries)
    - From *imodel* key to CDM value: can be done via transforming functions
    - From *imodel* value to CDM key is not likely initially, but can be done via transforming functions

# Use, input, output and syntax

## Input and output

The *cdm* tool maps the data elements stored in a *python* *pandas* *DataFrame* object to the CDM.

Input to the tool is:

1. A *python pandas DataFrame* object (or *pandas.io.parsers.TextFileReader* object) with data values in columns. Columns can be simple or multiindexed, to accommodate models with data in multiple sections and/or combination of data models.
2. A *python dictionary* with attributes of the elements in the input data. Currently, used attributes are (…)

Explain here that the data model must be mapped in the tool to the CDM

The output of the mapping is a *python* dictionary with the {*cdm\_table\_name*:*cdm\_table\_object*} pairs, that can be afterwards printed to ascii files with submodule xxxx. Where:

* *cdm\_table\_name:* name of the CDM table (i.e. header, observations\_at,….)
* *cdm\_table\_object*: *python* dictionary with the {*data*:*cdm\_table\_object,atts*:*cdm\_table\_atts*} pairs, where:
  + *cdm\_table\_object* is a *python pandas DataFrame* object with the CDM elements aligned in columns. Where:
    - The columns are the CDM data elements for which a mapping from the *imodel* is defined (there are no empty or null columns for CDM data elements not defined)
    - The CDM DataFrame index preserves the indexing of the input data
  + *cdm\_table\_atts*: python dictionary with the CDM element attributes, of which ??? *column\_type* and *decimal\_places* (inherited from the *imodel* mapping) are used to print the CDM table files.

CDM tables can be afterwards printed to an ascii files ready for insertion in the database with the xxxx submodule. The resulting set of files have the following characteristics:

* One ascii file per CDM table.
* One header line with column names.
* All columns defined in the CDM table are printed even if all records are null or not defined in the *imodel* mapping.
* observations\_\* tables have records with no observed value removed.

## Syntax

The calling sequence to map a model to the CDM from a python session is:

cdm\_dict = cdm.map\_model(imodel, data, atts, cdm\_subset = None, log\_level = ’INFO’)

List of arguments:

***imodel:*** name of imodel as registered in the tool (see ). Type: string.

***data:*** input data to map. Valid data types are: *pandas* DataFrame or io.parsers.TextFileReader objects or in-memory text streams (io.StringIO object). Type: string.

***atts:*** dictionary with the {element\_name:element\_attributes} of ***data***. Type: string.

List of keyword arguments:

***cdm\_subset:*** subset of CDM model tables to map. Defaults to the full set of CDM tables defined for the *imodel*. Type: list.

***log\_level:*** Defaults to ‘DEBUG’. Type: string.

The calling sequence to print the CDM table objects to ascii files is:

cdm.cdm\_to\_ascii(cdm\_dict, delimiter = ‘|’, extension = ‘psv’, null\_label = ‘null’, out\_dir = None, suffix = None, prefix = None, log\_level = ’INFO’)

List of arguments:

***cdm\_dict:*** name of imodel as registered in the tool (see ). Type: string.

List of keyword arguments:

***delimiter:*** field delimiter. Defaults to pipe. Type: string.

***extension:*** file extension. Defaults to ‘psv’. Type: string.

***null\_label:*** label for missing values. Defaults to ‘null’. Type: string.

***out\_dir:*** output directory. Type: string.

***suffix:*** suffix to add to table name to create the output filename. Type: string.

***prefix:*** prefic to add to table name to create the output filename. Type: string.

***log\_level:*** Defaults to ‘INFO’. Type: string.

## Combining data from different data models

In some cases, it is desired that the final CDM set of tables is composed of a combination of different data models/sources. Based on the IMMA1 reprocessing experience so far, this can be the case of:

1. Using a supplemental data element instead of the corresponding element in the IMMA1 core or attachments (i.e. to improve the final product by rescuing higher quality/precision supplemental data)
2. Adding data elements from a different data source (like adding WMO PUB 47 metadata)

For the first case, it is recommended that the input DataFrame and attributes have both data sources (the main-IMMA1 and supplemental) and that a mapping set (*imodel*) is defined specifically for that main-supplemental combination. This is to avoid the burden of:

* dependencies on element substitutions (ie. latitude, location\_accuracy)
* also to avoid the uncomfortable column replacement process when the input/output DataFrames are *pandas.io.parsers.TextFileReader* objects rather than plain DataFrames

If this approach is not possible or desired, separate mapping from different data models to the CDM can be done and column replacement performed afterwards, before printing the tables to ascii files. This is possible if both initial data sources have the same indexing, since indexing is preserved in the output CDM DataFrame objects before printing to ascii files, and thus column replacement can be done directly (although iterating over chunks if input/output DataFrames are *pandas.io.parsers.TextFileReader* objects)

For the second case, where data sources are completely different in nature ( IMMA1 individual reports Vs WMO PUB 47 station metadata), it is recommended to map both things separately and then make the appropriate replacements/additions based on the corresponding CDM element matching (i.e. ‘primary\_station\_id’). There should be a tool in the *common* module of *cdm* to do this.

# How to register a new data model mapping

Using IMMA1 as a reference, a data model (*imodel*) mapping can be understood in this context as:

1. A generic mapping from a defined data model, like IMMA1’s core and attachments.

i.e. *imodel* = imma1

1. A specific mapping from generic data model to CDM: map a SID-DCK from IMMA1’s core and attachments to CDM in a specific way.

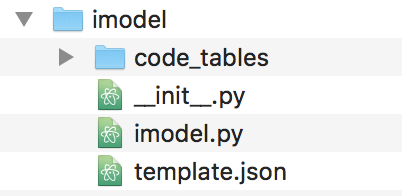
i.e. *imodel* = imma1\_*sid-dck*

1. A combination of multiple (2…?) data models in a single CDM mapping: map IMMA1’s core and attachments and use supplemental data for certain elements.

i.e. *imodel* = imma1\_*supp\_model\_name*

To perform any of these mappings to the CDM, the *imodel* mapping has to be included in the tool, by creating the *imodel* mapping library with the following steps:

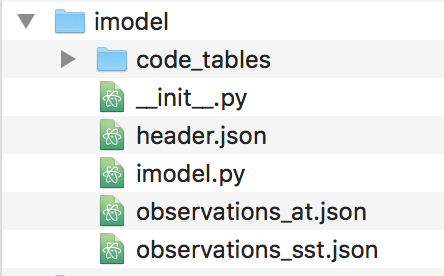
1. Copy the template mapping structure in the mappings’ library of the mapper module (./mapper/lib/), renaming the directory name and the template.py module to *imodel*.



The *imodel*.py module hosts the ﻿mapping\_functions class. Functions to map *imodel* elements to CDM elements (if required), have to be defined under this class for the tool to be able to access them. Additionally, the \_\_init\_\_.py file will make python recognize the *imodel* directory as containing modules, and hence being able to load it.

1. Create a copy of the template.json file (template mappings’ definition file) for each of the CDM tables *imodel* maps to. To access the CDM tables available for mapping through this tool type:

> table\_list = properties.cdm\_tables



1. Edit the mapping files (\*.json) and create the mappings to CDM elements from *imodel*:
   1. Details on how to define mappings to CDM elements are given in section 6.
   2. A dictionary with the names and pseudo-SQL types of the elements of each of the tables can be accessed typing:

> tables = cdm.cdm\_tables.table\_hdlr.load\_tables()

* 1. CDM element definitions can be found at? (link SW version with CDM version: cdm\_latest.pdf should not be,???!, if we live that in the tool, is it kind of dirty?/docs)
  2. If a CDM element is not mappable from the *imodel*, it need not be defined as blank in the mapping file, it can just be ignored.

1. Code tables!

# Data model mappings

Mapping a data model (*imodel*) to the elements in the CDM tables can be achieved using one of the following approaches:

1. Direct mapping from an *imodel* element
2. Direct mapping via code tables that can take one or multiple *imodel* elements
3. Assignment of a *default* value
4. Any other mapping including any combination of the following can be achieved using functions defined in the *imodel*.py module:
   1. *imodel* element’s attributes
   2. parameterization with input keyword arguments
   3. one or multiple *imodel* elements
   4. transforming function from/to coded elements other than direct mapping with code tables can be defined here (i.e.: *imodel* key to CDM value or *imodel* value to CDM key)
5. Circular dependencies (CDM element to CDM element) are currently not supported

The *imodel* mapping to the CDM must be defined in the *imodel* mapping library in individual mapping files (one per CDM table).

The following sections detail the mapping sequence the mapper follows to map to a CDM element and the mapping files and descriptors it uses to do so.

## CDM tables mapping files and descriptors

* **elements:** string or list of strings with element names in the data model to use in the mapping of the CDM element.
* ***sections:*** string or list of strings with sections the elements in ***elements*** (if the imodel data is partitioned in sections). Use a single string to define a unique section if all the elements in elements are located in the same section, use a list of strings in the same order as elements otherwise.
* **default:** value to assign to the CDM element. Type: any (datetime objects not supported….)
* **fill\_value:** value to assign to the CDM element in records where mapping has resulted in a NA/NaN value (missing data or no mapping…). Type: any (datetime objects not supported….)
* **transform:** name of the function in the mapping\_functions class of the *imodel*.py module to use to perform the mapping. Type: string
* **kwargs:** keyword arguments of function ***transform*** if any. Type: dictionary {‘keyword”:value,…,}
* **code\_table:** code table name in the *imodel* mapping library to use to perform the mapping. Type: string.
* **decimal\_places:** number of decimal places to keep on element printing. Can be either: integer value, function name used to estimate this figure (same as ***transform***). Currently, the functions defined under decimal\_places cannot take keyword arguments.

**Need to define as cero for integer elements defined as numeric in CDM or will be printed with default number of decimal places!**!!!

## Mapping sequence

The mapper parses the mapping file element by element and takes the following steps:

1. Clean *imodel* data. If ***elements*** is defined, reconvert to imodel elements to its original data type and remove missing. This preliminar step makes the definition of mapping functions easier, as no NaN handlings needs to be done there and integer fields casted to float by NA/NaN presence is reverted.
2. Map CDM element, in the following order:
3. If ***transform***: eval function and apply with ***elements*** and|or ***kwargs*** as appropiate
4. Else if ***code\_table***: map *imodel* ***elements*** using the defined ***code\_table***
5. Else if ***elements***: assign *imodel* ***elements*** to CDM element
6. Else if ***value***: assign ***value*** to CDM element
7. Fill CDM element NA/NaN values using ***default*** if defined.
8. Define the number of decimal places in the CDM element attributes to pass to the table writer if ***decimal\_places*** is provided.

## Defining mapping functions

General rules to apply:

All kind of mappings:

mapper only passes to transforming function or code table mapping rows where none of the elements implied are missing

Need to return np arrays or pd.Series, not pd.DataFrames. …or scalars

we need to define the internal data type of code table values:

* from imodel data they come as keys
* integers in cdm table definition in GitHub
* They are only integer values as code tables are defined in cdm: check that. Our need in the imodel to have them as key comes, partially, because they are defined in json files and keys are strings there.

Here we extract the cdm keys from the code table mapping as the value field of the key:value pair: imodel\_key:cdm\_key, and thus can be comfortably fit in an int dtype definition.

!Internally, the cdm code tables are transformed and treated as integers

output must be of the same kind as cdm element definition:

if a cdm element is defined in the GitHub table\_definition as numeric and is int, it will be printed with default decimal places unless decimal\_places = 0!!!!!

* varchar[],numeric[],int[] will be arrays, list of the corresponding type, pd column dtype will be an object…

When assigning arrays: have to double-enclose to give appropriate dimensionality

1. value, default: [1,2,3] in mapping file (will do the job internally)
2. when building mapping functions remember: [[]]\*len(df.index)

* varchar: str, pd type will be object
* int: int (float internally if any NaN in column)
* numeric!: float
* datetimes?????: datetime probably!

Missing inside arrays? if this is potentially possible, to nan in df map. When writing to ascii df table:

* remove NaN element from array
* on empty array: NULL or {}?

decimal\_places needs to be another attribute in the element mapping of floats:

* If it is not defined, just let python work freely, no internal defaults?
* How to define an elememet decimal places:
  + either from imodel atts (via function)
  + from functions (depending on element transformation, etc….)
  + fixed value
* So far, the function to calculate decimal places takes a an only argument: the imodel data element(s) ( always as a list) used to create the cdm element
  + When is directly from schema: general func, an only takes 1 value list[0]
  + Any other type of decimal places estimation will have its function defined in the mappings….

## Direct code table keys mapping

The *schema* file gathers a collection of descriptors that enable the mdf\_reader to access the content of a data format and to extract the data it contains to meaningful units of information.

Mapping tables in: mapper/lib/*imodel*/code\_tables

Format (& extension): json

Name: short & meaningfull…., probably just cdm\_key name, but don’t make the code code get the cdm\_key as the codetable name, treat them separatelly

# CDM tables definition

Ref: GitHub/glamod/common\_data\_model/table\_definitions

Path: /tables/lib

Files: header.json, observations.json

Content: cdm\_key:{‘column\_type’:python\_language + str, object, key, datetime, so far….}

Internally updated to the following in numeric float fields to pass to the table writer:

cdm\_key:{‘column\_type’:python\_language, ‘decimal\_places’:int}

Also internally, in the mapped object, the column\_type may differ from the type declared here, as NaN will make int fields to be promoted to float. This will not be updated in the table att dictionarty above in the mapper, a different interal

## Updating tables from GLAMOD Common Data Model repository

tables.table\_hdlr.from\_glamod(gitlink)

Will return a dictionary with key:{‘column\_type’:value}

Optionally directly to file:

﻿with open('/Users/iregon/test.json','w') as fileObj:

json.dump(tables.table\_hdlr.from\_glamod(gitlink),fileObj,indent=4)

# Replacing mappings

If a data source (data coming from a specific data model) wants to be improved with data coming from a different data source (i.e. add MD from pub47 or replace data from supplemental):

1. We map both sources of data to the CDM
2. Then we apply substitutions on the CDM space

Some things to keep in mind:

* Replacements should be enabled to occur on the index or on a value (map a record id to a sensor type).
* If on the index (like substituting easily supp data): mapping to the CDM needs to keep the original indexing of the input data in both sources of data (this is probably the default behavior of pandas) and also all the records (if we remove empty ones and work with parsers, we might not find the same data in the same chunk….). Removal of empty data will be only performed when tables are written to ascii
* To do this in a clean way, we create a module with df management in the common module.