POTSDAM-INSTITUT FÜR
KLIMAFOLGENFORSCHUNG

Geosci. Model Dev., 8, 1877–1883, 2015
www.geosci-model-dev.net/8/1877/2015/
doi:10.5194/gmd-8-1877-2015

Geoscientific Model Development
Open Access

NCIO 1.0: a simple Fortran NetCDF interface

A. Robinson[1,2,3] **and M. Perrette**[3]

[1]Universidad Complutense de Madrid, 28040 Madrid, Spain
[2]Instituto de Geociencias, UCM-CSIC, 28040 Madrid, Spain
[3]Potsdam Institute for Climate Impact Research, 14473 Potsdam, Germany

Correspondence to: A. Robinson (robinson@fis.ucm.es)

Received: 19 November 2014 – Published in Geosci. Model Dev. Discuss.: 16 January 2015
Revised: 10 June 2015 – Accepted: 16 June 2015 – Published: 30 June 2015

Abstract. The NetCDF (Network Common Data Form) library has become an indispensable tool for data and model output management in geoscience. However for simple tasks, particularly in Fortran, the complexity of native NetCDF functionality can be cumbersome. The NCIO (NetCDF Input/Output) module has been designed as an interface to the NetCDF library with simplicity and ease of use in mind. While this implies that some NetCDF functionality is masked from the user, the subroutines provided here are adequate for basic serial reading and writing tasks of up to 6-D data arrays along with corresponding data attributes. The code is available online via a GitHub repository (http://www.github.com/alex-robinson/ncio), which includes an example program to illustrate the approach.

1 Introduction

The NetCDF (Network Common Data Form) library developed by Unidata (Unidata, 2014) has revolutionized the storage of large geoscientific data sets, reproducibility of experiments and archiving of model output. It eliminates the dependencies of binary output on a given computing system and ensures that data is fully self-described within a file. The fact that its use has been adopted so widely in the geoscientific community has ensured that many tools are available for data processing and analysis based on this storage format.

To provide a library that is useful over a wide range of applications, the low-level functionality of NetCDF gives users full control over how a program interacts with the data files. For this reason, the native NetCDF interface relies on a series of intermediate function calls and helper variables to be able to read or write data. This flexibility can be critical for some applications, for example for storing large, complex data sets with many attributes or for parallel I/O (input/output) in global circulation models (GCMs; Huang et al., 2014).

However, for more common tasks, these intermediate steps tend to make programming data I/O with NetCDF more complex and even cumbersome, in some cases. Several versions of NetCDF wrappers already exist to make NetCDF writing more straightforward, such as Gtool5 (Ishiwatari et al., 2012) meant for use in GCMs, or the Climate Model Output Rewriter (CMOR, 2015), which aides in writing climate and forecast (CF, 2015) compliant files. These libraries share common characteristics in that they mask some of the intermediate steps of loading and writing NetCDF data from the user. However, at least for Fortran, a NetCDF wrapper that was both generic and simple has not been available until now.

The NCIO (NetCDF Input/Output) module is intended to fill that gap. The goal of the module is to provide access to NetCDF functionality in Fortran in the simplest way possible. This implies that it is not appropriate for every application, but it should be generic enough to be widely useful. Here NCIO is described and provided, in the hope that it will be helpful for others developing geoscientific models. The application programming interface (API) is provided in the Appendix.

2 Reading from NetCDF

NCIO contains one subroutine for reading data from NetCDF files: `nc_read`. This subroutine can handle the reading of scalars, vectors and arrays of up to 6 dimensions. The data to be read can be of type integer, float, double precision or logical. The subroutine has an internal module interface for

each data type and data size, so it automatically handles any of the above inputs from the user.

In its most basic form, the user supplies the filename to be read from, the name of the variable to read and the scalar or array that will hold the data. In this case, the size of the variable in the file and the dimensions of the array should be consistent. The user can also optionally supply `start` and `count` vectors, which indicate which slices of the array should be read from the file (analogous to low-level NetCDF functions). If neither is given, the subroutine will start from the beginning of the data record in each dimension and count up to the size of the array in each dimension.

`nc_read` can handle missing data, as long as the variable attribute `missing_value` is defined in the NetCDF file. The user can specify a new value to represent missing data via the `nc_read` subroutine call. NCIO reads the data from the file and replaces missing data points with the desired value.

In order to facilitate the reading of data of unknown dimension sizes, an additional helper function `nc_size` is available. This function returns the size of a dimension variable in the NetCDF file. This can be used, for example, to determine the dimensions of the array to allocate in Fortran before reading data from the NetCDF file.

Additionally, it is possible to read string attributes from the file using `nc_read_attr`. The user provides the filename, the name of the attribute of interest and optionally the pertinent variable name (if it is not a global attribute), and the subroutine will return the string associated with that attribute in the file.

3 Writing to NetCDF

Writing NetCDF files is achieved through one subroutine for writing variables (`nc_write`), two necessary subroutines for file initialization (`nc_create` and `nc_write_dim`), and two optional ones for adding attributes (`nc_write_attr` and `nc_write_map`). The data to be written to a file, as with `nc_read`, can be a scalar, vector, or an array of up to 6 dimensions and of type integer, float, double precision or logical. The subroutine has an internal module interface for each data type and data size, so it automatically handles any of the above inputs from the user.

The subroutine `nc_create` is used to initialize the new NetCDF file. It does nothing more than open a new file, optionally write a few typical global attributes and close the file. This leaves an empty NetCDF file available to be filled with dimensions and data. By default this function will overwrite any previous file with the same name, and it will write in the classic NetCDF format. After creating the file, the user can optionally write additional global attributes to the file with `nc_write_attr`. This subroutine is useful for specifying references and data set information, for example.

It is also possible to specify a grid mapping variable in the file via `nc_write_map`, which can be used to define a projection or other type of map to associate with a grid variable. Such a map definition is helpful to (and sometimes necessary for) programs that plot NetCDF data, such as Panoply (Schmunk, 2014), when the spatial coordinates of the data are not longitude and latitude. Currently, only stereographic and polar stereographic projections are handled by this subroutine, but it is planned to be made more generic in the future.

Once the NetCDF file is prepared with global information, the next step is to define dimension variables, which is done with the subroutine `nc_write_dim`. To write a dimension variable, the user must supply the filename and name of the variable to be written. The vector of dimension values can be provided directly, or a vector of values can be generated from a starting value, the total number of points and the distance between them. When the argument `unlimited=.TRUE.` is specified, then the dimension can be extended via calls to `nc_write` (see below). Additionally, some Climate and Forecast (CF) variable attributes can be supplied via this subroutine, such as `long_name` and `units`, while the subroutine `nc_write_attr` can be used directly to write any additional variable specific attributes. It should be noted that all dimension variables must be written to the file before any data can be written.

The NetCDF file should now be ready for storing data. The initialization of a new variable in the file, as well as writing of the data, is achieved via the subroutine `nc_write`. To write a variable to the file, the user must supply the filename and name of the variable to be written, along with the data array and the names of the dimensions to be associated with the variable. The dimension variables should already exist in the file or an exception will be thrown. Additional variable attributes can be written if desired. Similar to `nc_read`, the user can optionally specify `start` and `count` vectors to write the data to a specific slice of the variable in the file. `nc_write` will check if the variable exists in the file and, if not, the variable will be created with the specified characteristics. In future calls to the subroutine, only the data will be written to the file.

4 Discussion

The NCIO module is intended to provide a clean and direct interface to the reading and writing of NetCDF files in Fortran. All NCIO functionality is contained in one portable module file (ncio.f90). No additional configuration is needed for its use, other than a `use ncio` statement in the Fortran program (replacing any `use netcdf` statements). Error handling follows the native NetCDF error protocol to maintain transparency. The simplicity of this approach facilitates the use of the NetCDF data format even for rather small programs or during prototyping. In other words, the goal is

to make reading and writing NetCDF files as easy as reading and writing binary or ASCII data. The subroutine design in NCIO mimics what is available in other languages, such as the ncdf library in R (Pierce, 2014), and should be useful for a wide range of applications. The basic functionality provided here has been tested thoroughly and works robustly in several real geoscientific programs.

The subroutines have been designed to mask any intermediate and temporary variables from the user. For example, low-level NetCDF functions make use of ID values of the file and other variables to know what is being loaded or written. Here only the filename and name of the variable are needed to find the right field in the NetCDF file, because the low-level functions are wrapped by the NCIO subroutines. In this way, the only variable that needs to be defined in the user's program is the data itself. Variable initialization and attribute definition, as well as data writing, are all easily handled with one subroutine call. Thus, for quick diagnostic output to a NetCDF file, for example, a user can easily write the variable to a file without additional variable definitions or subroutine calls. For example, Unidata provides a NetCDF tutorial program to write 4-D temperature and pressure fields (pres_temp_4D_wr.f90 found at http://www.unidata.ucar.edu/software/netcdf/docs/netcdf-tutorial). The native NetCDF approach require 25 lines of code and 25 intermediate variable definitions, while NCIO achieves the same result with only 7 lines of code.

Such ease of use does incur minor computational costs. One such cost comes from the overhead of opening and closing the NetCDF file within each subroutine call, rather than leaving the NetCDF file open for writing throughout the entire program. The reason for this approach was to maintain the simplest interface possible – otherwise additional functions and external variables would be needed for opening and closing the file. In testing the NCIO module, it was found that in most cases, any additional overhead incurred is small in absolute terms and does not appear to affect the speed of a program significantly. Comparing the native and NCIO-based tutorial program pres_temp_4D_wr.f90, we find that one iteration of writing these variables takes about 4 times longer using NCIO. For 1000 iterations, using NCIO takes about 80 times longer than the native NetCDF calls. However, in absolute terms, the total time for 1000 iterations using NCIO is about 1 s.

Nonetheless, for programs that write data with a very high frequency, this approach could result in noticeably slower execution than using the low-level NetCDF functions directly. For this reason, all reading/writing subroutines include an optional "ncid" argument and two additional subroutines are provided in the library: `nc_open` and `nc_close`. These subroutines allow the user to keep a given NetCDF file through multiple NCIO subroutine calls and identify the open file using the "ncid" intermediate variable. Using this approach reduces the cost of 1 and 1000 iterations as above to about 2 and 20 times longer than the native calls, respectively. It should be noted that the timing is dependent on the size of the arrays being written as well. The default test is for rather small arrays ($12 \times 6 \times 2$ grid points). When writing larger arrays (e.g., $100 \times 100 \times 2$ grid points), NCIO consistently needs about 10 times longer than the native calls using either approach.

5 Conclusions

NCIO is intended to provide an easy-to-use interface to the NetCDF library in Fortran. At the cost of more complex native NetCDF functionality, a minimal set of subroutines was developed to handle the most common reading and writing tasks of up to 6-D arrays. The functions available to the user have been described here, such that simply downloading and compiling the module will allow the user to immediately begin using it.

Appendix A

Table A1. Subroutine call and argument descriptions for `nc_read`.

`subroutine nc_read(filename,name,dat,[start],[count],[missing_value],[ncid])`	
`filename`	name of the NetCDF data file to read from
`name`	name of the variable in the NetCDF file to be read
`dat`	output: Fortran data type into which data will be loaded
`start`	vector of values specifying starting indices for reading data from each dimension (optional)
`count`	vector of values specifying how many values to read in each dimension (optional)
`missing_value`	value to assign to missing data read from the file (optional)
`ncid`	file ID for a file that remains open for various NCIO calls (optional)

Table A2. Function call and argument descriptions for `nc_size`.

`function nc_size(filename,name,[ncid]) result(size)`	
`filename`	name of the NetCDF data file to read from
`name`	name of the dimension variable in the NetCDF file of which to determine size
`size`	output: integer size (length) of the dimension variable returned from the function
`ncid`	file ID for a file that remains open for various NCIO calls (optional)

Table A3. Subroutine call and argument descriptions for `nc_read_attr`.

`subroutine nc_read_attr(filename,[varname],name,value,[ncid])`	
`filename`	name of the NetCDF file from which to read attribute
`varname`	name of the variable from which to read the attribute (optional)
`name`	name of the attribute to be read
`value`	output: value of the attribute to be read
`ncid`	file ID for a file that remains open for various NCIO calls (optional)

Table A4. Subroutine call and argument descriptions for `nc_create`.

`subroutine nc_create(filename,[overwrite],[netcdf4],[author],[creation_date],[institution],[description])`	
`filename`	name of the NetCDF file to be created
`overwrite`	switch to determine whether file on disk should be overwritten (optional, default TRUE)
`netcdf4`	switch to determine whether file format should be NetCDF4 (optional, default FALSE)
`author`	name of the author of the file (optional)
`creation_date`	date of the file creation, string format (optional)
`institution`	name of the author's institution (optional)

Table A5. Subroutine call and argument descriptions for `nc_write_attr`.

`subroutine nc_write_attr(filename,[varname],name,value,[ncid])`	
`filename`	name of the NetCDF file in which to write attribute
`varname`	name of the variable to which the attribute should be associated (optional)
`name`	name of the attribute to be written
`value`	value of the attribute to be written
`ncid`	file ID for a file that remains open for various NCIO calls (optional)

Table A6. Subroutine call and argument descriptions for `nc_write_map`.

`subroutine nc_write_map(filename,name,[lambda],[phi],[x_e],[y_n],[ncid])`	
`filename`	name of the NetCDF file in which to write the grid map definition
`name`	name of the grid mapping to be defined
`lambda`	longitude of projection origin (optional)
`phi`	latitude of projection origin (optional)
`x_e`	false easting (optional)
`y_n`	false northing (optional)
`ncid`	file ID for a file that remains open for various NCIO calls (optional)

Table A7. Subroutine call and argument descriptions for `nc_write_dim`.

`subroutine nc_write_dim(filename,name,[x],[dx],[nx],[long_name], [standard_name],[units],[axis],[calendar],[unlimited],[ncid])`	
`filename`	name of the NetCDF file in which to define dimension
`name`	name of the dimension to be defined in NetCDF file
`x`	Fortran data type (scalar or vector) specifying values of dimension. If `nx` is present and size(`x`)==1, `x` specifies the starting point of the dimension variable
`dx`	distance between each dimension value (optional)
`nx`	length of dimension variable (optional)
`long_name`	NetCDF attribute, a long descriptive name of the variable (optional)
`standard_name`	NetCDF attribute specifying the CF convention standard name of the variable (optional)
`units`	NetCDF attribute of the units of the variable (optional)
`axis`	NetCDF attribute of the standard axis of the variable (optional)
`calendar`	NetCDF attribute of the calendar type to be used for time dimensions (optional)
`unlimited`	NetCDF attribute to determine whether the dimension can be extended after its initial definition or not
`ncid`	file ID for a file that remains open for various NCIO calls (optional)

Table A8. Subroutine call and argument descriptions for `nc_write`.

`subroutine nc_write(filename,name,dat,[dims],[dim1,...,dim6],[start],[count], [long_name],[standard_name],[grid_mapping],[units],[missing_value],[ncid])`	
`filename`	name of the NetCDF file in which to write data
`name`	name of the variable in NetCDF file to be written
`dat`	data to be written
`dims`	vector of dimension names of the variable in the NetCDF file (optional)
`dim1,...,dim6`	individual dimension names of the variable in the NetCDF file (optional)
`start`	vector of values specifying starting indices for reading data from each dimension (optional)
`count`	vector of values specifying how many values to read in each dimension (optional)
`long_name`	NetCDF attribute, a long descriptive name of the variable (optional)
`standard_name`	NetCDF attribute specifying the CF convention standard name of the variable (optional)
`grid_mapping`	name of the grid this variable is mapped on (optional)
`units`	NetCDF attribute of the units of the variable (optional)
`missing_value`	value of missing data to be written to file (optional)
`ncid`	file ID for a file that remains open for various NCIO calls (optional)

Table A9. Subroutine call and argument descriptions for `nc_open`.

`subroutine nc_open(filename,ncid,[writable])`	
`filename`	name of the NetCDF file from which to read attribute
`ncid`	output: integer variable to identify a NetCDF file through multiple NCIO calls
`writable`	switch to determine whether file should be opened for writing (optional, default TRUE)

Table A10. Subroutine call and argument descriptions for `nc_close`.

`subroutine nc_close(ncid)`	
`ncid`	integer variable to identify a NetCDF file to be closed

Code availability

The NCIO module is open source and available under the MIT License, making it suitable for community-driven development or for each user to adapt the module to more particular needs. The code is hosted in a public Git repository located at http://www.github.com/alex-robinson/ncio. Suggestions and improvements are welcome. Any GitHub users who wish to make a contribution should email the authors to add them as contributors. The above-mentioned tutorial comparison is included in the repository, as well as a simple test program with its output to ensure the code is working properly and to provide examples of subroutine calls.

Acknowledgements. We would like to thank Mario Krapp and Reinhard Calov for valuable input and testing of the NCIO module. A. Robinson is supported by the Marie Curie 7th framework programme (project 2012-IEF-331835, EURICE).

Edited by: D. Roche

References

CF: Climate and Forecast Conventions and Metadata, available at: http://cfconventions.org/ (last access: 25 June 2015), 2015.

CMOR: Climate Model Output Rewriter, available at: http://pcmdi.github.io/cmor-site/index.html (last access: 25 June 2015), 2015.

Huang, X. M., Wang, W. C., Fu, H. H., Yang, G. W., Wang, B., and Zhang, C.: A fast input/output library for high-resolution climate models, Geosci. Model Dev., 7, 93–103, doi:10.5194/gmd-7-93-2014, 2014.

Ishiwatari, M., Toyoda, E., Morikawa, Y., Takehiro, S., Sasaki, Y., Nishizawa, S., Odaka, M., Otobe, N., Takahashi, Y. O., Nakajima, K., Horinouchi, T., Shiotani, M., Hayashi, Y.-Y., and Gtool development group: "Gtool5": a Fortran90 library of input/output interfaces for self-descriptive multi-dimensional numerical data, Geosci. Model Dev., 5, 449–455, doi:10.5194/gmd-5-449-2012, 2012.

Pierce, D.: ncdf: Interface to Unidata netCDF data files, available at: http://CRAN.R-project.org/package=ncdf (last access: 25 June 2015), r package version 1.6.8, 2014.

Schmunk, R. B.: Panoply, available at: http://www.giss.nasa.gov/tools/panoply/ (last access: 25 June 2015), 2014.

Unidata: NetCDF, available at: http://www.unidata.ucar.edu/software/netcdf/ (last access: 25 June 2015), 2014.