Introduction to using the netCDF data format with Fortran 90

Michael Thorne (michael.thorne@utah.edu)
Last Updated: July 20, 2010

I. Introduction

NetCDF – **Net**work **C**ommon **D**ata **F**orm

NetCDF is an array based data structure for storing multidimensional data. A netCDF file is written with an ASCII header and stores the data in a binary format. The space saved by writing binary files is an obvious advantage, particularly because one does not need worry about the byte order. Any byte-swapping is automatically handled by the netCDF libraries and a netCDF binary file can thus be read on any platform. Some features associated with using the netCDF data format are as follows:

Coordinate systems: support for *N*-dimensional coordinate systems.

- X-coordinate (e.g., lat)
- Y-coordinate (e.g., lon)
- Z-coordinate (e.g., elevation)
- Time dimension
- Other dimensions

Variables: Support for multiple variables.

- E.g., *S*-wave velocity, *P*-wave velocity, density, stress components…

Geometry: Support for a variety of grid types (implicit or explicit).

- Regular grid (implicit)
- Irregular grid
- Points

Self-Describing: Dataset can include information defining the data it contains.

- Units (e.g., km, m/sec, gm/cm^3,…)
- Comments (e.g., titles, conventions used, names of variables (e.g., P-wave velocity), names of coordinates (e.g., km/sec),...

Full documentation on the data format can be found at:

- http://www.unidata.ucar.edu/software/netcdf/ - netCDF webpage
- http://www.unidata.ucar.edu/software/netcdf/docs/ - Full netCDF documentation
- http://www.unidata.ucar.edu/software/netcdf/docs/netcdf-f90/ - F90 Interface guide

Naming convention: NetCDF files are generally named with the *.nc* extension.

File conventions: There are many different flavors or conventions of netCDF data. In seismology the example most people are familiar with are the *.grd* files produced with the Generic Mapping Tools (http://gmt.soest.hawaii.edu/). Writing netCDF files to use with GMT requires using the COARDS ("Cooperative Ocean/Atmosphere Research Data Service") data convention. The specifics of this convention can be found at: http://ferret.wrc.noaa.gov/noaa_coop/coop_cdf_profile.html.

One of the specific reasons I have started using netCDF files is that the grid information can also be stored along with the data. Hence when using a program like GMT I don't explicitly have to write an *xyz2grd* command to plot the data or try to remember for the particular file I am plotting what the Range (**-R** option in GMT) or grid increment (**-I** option) are. However, when using the COARDS convention one is limited to rectilinear coordinate systems. The coordinates do not have to be evenly spaced, but must at least be either monotonically increasing or decreasing. Unfortunately GMT cannot handle irregularly spaced grids without preconditioning the data with a command like *blockmean*.

Visualization: Another obvious advantage of using the netCDF format is that a variety of data viewers already exist. Additionally nearly all visualization packages that I am familiar with read the netCDF format. Some popular free viewers or visualization packages are:

- Ncview - http://meteora.ucsd.edu/~pierce/ncview_home_page.html
- Data Explorer (OpenDX) - http://www.opendx.org/

II. File Structure

A netCDF file contains the following structures.

NetCDF file contains:	
Global Attributes:	Describe the contents of the file.
Dimensions:	Define the structure of the data (e.g., time, depth, lat, lon).
Variables:	Holds the data in arrays shaped by dimensions
Variable Attributes:	Describes the content of each variable.

III. Compiling and Linking

The examples shown are using the flags available with the g95 compiler. In order to use netCDF with f90 you need to have the f90 netcdf module installed. This module should be located in your netCDF installation directory under: *include/netcdf.mod*. If this module isn't there you may need to reinstall netCDF.

To compile a module with subroutines that uses netcdf calls. A possible module file will have the following setup:

```
MODULE example
CONTAINS

      SUBROUTINE sub1( )
      USE netcdf
      IMPLICIT NONE
      ...
      END SUBROUTINE sub1
      ...
END MODULE example
```

As shown in this example, you must make the netCDF subroutines available to your subroutine by the *USE netcdf* statement.

If this module is called example.f90 then it can be compiled by:

```
>> g95 -c example.f90 -I $(NETCDF)/include
```

Where $(NETCDF) is a shell variable providing the path to the netCDF installation. The -I option in g95 tells the compiler where to look for the *netcdf.mod* module.

Compiling and linking a program (e.g., 'main.f90') to the pre-compiled *example.o* module from above is then done by:

```
>> g95 main.f90 -o main.x ./example.o -L$(NETCDF)/lib -lnetcdf
```

Where the **-L** and **-l** options must be specified (for g95 - see compiler specific flags) if the netCDF libraries are not already in your path.

IV. Utilities

NetCDF comes with a couple of useful utilities that should be located in the *bin/* directory of your netCDF installation.

ncdump: generate an ASCII representation of a specified netCDF file to standard output.

To just show the header information:

```
>> ncdump –h filename.nc
```

ncgen: this can be used to generate a netCDF file.

V. Example - Reading a netCDF file with F90

To read a netCDF file into a f90 program when you have no information about the size or names of the variables, the order of the library calls to use is as follows:

```
NF90_OPEN                          ! open existing netCDF dataset
 NF90_INQUIRE                      ! find out what is in it
    NF90_INQUIRE_DIMENSION         ! get dimension names, lengths
    NF90_INQUIRE_VARIABLE          ! get variable names, types, shapes
      NF90_INQ_ATTNAME             ! get attribute names
      NF90_INQUIRE_ATTRIBUTE       ! get other attribute information
      NF90_GET_ATT                 ! get attribute values
    NF90_GET_VAR                   ! get values of variables
 NF90_CLOSE                        ! close netCDF dataset
```

As an example, consider reading a 2D GMT grid file into a F90 program. The first thing I want to do in my program is determine the size of the grid. I can accomplish this by making a call to the supplied subroutine griddims…

```
CALL griddims(infile,NX,NY)
```

Where *infile* is the name of the netCDF file I want to read in and *NX* and *NY* are integers that will hold the size of the grid file.

```
!GRIDDIMS - Get dimensions of a netCDF 2D gridfile
!:============================================================
SUBROUTINE griddims(infile,NX,NY)
USE netcdf
IMPLICIT NONE
INTEGER(KIND=4), INTENT(OUT) :: NX, NY
INTEGER(KIND=4) :: ncid
CHARACTER(LEN=50), INTENT(IN) :: infile
CHARACTER(LEN=50) :: xname, yname
```

```
!Open netCDF file
!:-------:-------:-------:-------:-------:-------:-------:
CALL check(nf90_open(infile, nf90_nowrite, ncid))

!Inquire about the dimensions
!:-------:-------:-------:-------:-------:-------:-------:
CALL check(nf90_inquire_dimension(ncid,1,xname,NX))
CALL check(nf90_inquire_dimension(ncid,2,yname,NY))

!Close netCDF file
!:-------:-------:-------:-------:-------:-------:-------:
CALL check(nf90_close(ncid))

END SUBROUTINE griddims
!:=====================================================================
```

Now that I have my grid dimensions I can allocate an array to hold the data:

```
ALLOCATE(indata(NX,NY))
ALLOCATE(xpos(NX))
ALLOCATE(ypos(NY))
```

Where I will put the data in the array *indata*, and the X- and Y- grid locations in the arrays *xpos* and *ypos* respectively. And now I can make a subroutine call to readgrid to get the data out of the netCDF file.

```
CALL readgrid(infile,xpos,ypos,indata,NX,NY)
```

Now you can do whatever you want with the netCDF data! Of special note though, the subroutine provided in this example is not completely generalized. One can write routines to read in completely generic netCDF files. In this example, I know that I am reading a 2D grid, and that the x-coordinate positions are written to the first variable, the y-coordinate positions are written to the second variable, and the array data is written to the third variable.

```
!READGRID - read a netCDF gridfile
!:=====================================================================
SUBROUTINE readgrid(infile,xpos,ypos,idata,NX,NY)
USE netcdf
IMPLICIT NONE
REAL(KIND=4), DIMENSION(NX), INTENT(OUT) :: xpos
REAL(KIND=4), DIMENSION(NY), INTENT(OUT) :: ypos
REAL(KIND=4), DIMENSION(NX,NY), INTENT(OUT) :: idata
INTEGER(KIND=4), INTENT(IN) :: NX, NY
INTEGER(KIND=4), DIMENSION(2) :: dimids
INTEGER(KIND=4) :: ncid, xtype, ndims, varid
CHARACTER(LEN=50), INTENT(IN) :: infile
CHARACTER(LEN=50) :: xname, yname, vname

!Open netCDF file
!:-------:-------:-------:-------:-------:-------:-------:-------:
CALL check(nf90_open(infile, nf90_nowrite, ncid))

!Get the values of the coordinates and put them in xpos & ypos
!:-------:-------:-------:-------:-------:-------:-------:-------:
CALL check(nf90_inquire_variable(ncid,1,vname,xtype,ndims,dimids))
CALL check(nf90_inq_varid(ncid,vname,varid))
CALL check(nf90_get_var(ncid,varid,xpos))

CALL check(nf90_inquire_variable(ncid,2,vname,xtype,ndims,dimids))
CALL check(nf90_inq_varid(ncid,vname,varid))
CALL check(nf90_get_var(ncid,varid,ypos))

!Get the values of the perturbations and put them in idata
!:-------:-------:-------:-------:-------:-------:-------:-------:
CALL check(nf90_inquire_variable(ncid,3,vname,xtype,ndims,dimids))
CALL check(nf90_inq_varid(ncid,vname,varid))
CALL check(nf90_get_var(ncid,varid,idata))

!Close netCDF file
!:-------:-------:-------:-------:-------:-------:-------:-------:
CALL check(nf90_close(ncid))

END SUBROUTINE readgrid
!:=====================================================================
```

The subroutine readgrid utilizes the subroutine **check**, which will return a netCDF error and stop execution of the code if netCDF encounters an error.

```
!Check (ever so slightly modified from www.unidata.ucar.edu)
!:==============================================================
SUBROUTINE check(istatus)
USE netcdf
IMPLICIT NONE
INTEGER, INTENT (IN) :: istatus
IF (istatus /= nf90_noerr) THEN
 write(*,*) TRIM(ADJUSTL(nf90_strerror(istatus)))
END IF
END SUBROUTINE check
!:==============================================================
```

VI. Example - Writing a netCDF file with F90

To write a netCDF file from a F90 program the order of the library calls to use is as follows:

```
NF90_CREATE           ! create netCDF dataset: enter define mode

   NF90_DEF_DIM       ! define dimensions: from name and length

   NF90_DEF_VAR       ! define variables: from name, type, dims

   NF90_PUT_ATT       ! assign attribute values

NF90_ENDDEF           ! end definitions: leave define mode

   NF90_PUT_VAR       ! provide values for variable

NF90_CLOSE            ! close: save new netCDF dataset
```

In the following example I show a simple way to write a 2D array of data into a netCDF file. Here is an example where I just make up a cosine variation on a globe:

```
PROGRAM ex
IMPLICIT NONE
REAL(KIND=4), DIMENSION(:,:), ALLOCATABLE :: mydata
REAL(KIND=4), DIMENSION(:), ALLOCATABLE :: xpos, ypos
REAL(KIND=4) :: x, y
INTEGER(KIND=4) :: NX
INTEGER(KIND=4) :: NY
INTEGER(KIND=4) :: J, K
CHARACTER(LEN=50) :: outfile

outfile = 'ex1.nc'
NX = 360
NY = 180

ALLOCATE(mydata(NX,NY))
ALLOCATE(xpos(NX))
ALLOCATE(ypos(NY))

!Populate X and Y grid locations
x = -179.0
DO J=1,NX
 xpos(J) = x
 x = x + 1.0
ENDDO

y = -90.0
DO J=1,NY
 ypos(J) = y
 y = y + 1.0
ENDDO

!make up some data
DO J=1,NX
  x = xpos(J)
  DO K=1,NY
    y = ypos(K)

    mydata(J,K) = cos(sqrt(x*x+y*y)/10.0)

  ENDDO
ENDDO

!write netCDF file
CALL writegrid(outfile,xpos,ypos,mydata,NX,NY)

END PROGRAM ex
```

This program uses the following *writegrid* subroutine:

```
!WRITEGRID - write a netCDF gridfile
!:============================================================================
SUBROUTINE writegrid(outfile,xpos,ypos,idata,NX,NY)
USE netcdf
IMPLICIT NONE
REAL(KIND=4), DIMENSION(NX), INTENT(IN) :: xpos
REAL(KIND=4), DIMENSION(NY), INTENT(IN) :: ypos
REAL(KIND=4), DIMENSION(NX,NY), INTENT(IN) :: idata
INTEGER(KIND=4) :: ncid, x_dimid, y_dimid
INTEGER(KIND=4) :: x_varid, y_varid, varid
INTEGER(KIND=4), DIMENSION(2) :: dimids
INTEGER(KIND=4), INTENT(IN) :: NX, NY
CHARACTER(LEN=50), INTENT(IN) :: outfile

!Create the netCDF file.
CALL check(nf90_create(outfile, NF90_CLOBBER, ncid))

!Define the dimensions.
CALL check(nf90_def_dim(ncid, "lon", NX, x_dimid))
CALL check(nf90_def_dim(ncid, "lat", NY, y_dimid))

!Define coordinate variables
CALL check(nf90_def_var(ncid, "lon", NF90_REAL, x_dimid, x_varid))
CALL check(nf90_def_var(ncid, "lat", NF90_REAL, y_dimid, y_varid))
dimids = (/ x_dimid, y_dimid /)

!Define variable
CALL check(nf90_def_var(ncid, "Perturbations", NF90_FLOAT, dimids, varid))
CALL check(nf90_enddef(ncid)) !End Definitions

!Write Data
CALL check(nf90_put_var(ncid, x_varid, xpos))
CALL check(nf90_put_var(ncid, y_varid, ypos))
CALL check(nf90_put_var(ncid, varid, idata))
CALL check(nf90_close(ncid))

END SUBROUTINE writegrid
!:============================================================================
```

VII. Example – Adding units and other attributes

Units and other attributes are easily added to a netCDF file. There are some standard attributes that can be added. A list of these standard attributes is given at:
http://www.unidata.ucar.edu/software/netcdf/docs/netcdf-f90/Attribute-Conventions.html#Attribute-Conventions

A couple of the most useful attributes are:

A few standard attributes:

long_name	A long descriptive name. This could be used for labeling plots, for example.
valid_min	A scalar specifying the minimum valid value for this variable.
valid_max	A scalar specifying the maximum valid value for this variable.
title	A global attribute that is a character array providing a succinct description of what is in the dataset.
history	A global attribute for an audit trail. This is a character array with a line for each invocation of a program that has modified the dataset.

Here I show a brief example of how to go about doing this. The first steps of opening a file and making the definitions are done as in the example of the previous section:

```
!Create the netCDF file.
CALL check(nf90_create(outfile, NF90_CLOBBER, ncid))

!Define the dimensions.
CALL check(nf90_def_dim(ncid, "lon", NX, x_dimid))
CALL check(nf90_def_dim(ncid, "lat", NY, y_dimid))

!Define coordinate variables
CALL check(nf90_def_var(ncid, "lon", NF90_REAL, x_dimid, x_varid))
CALL check(nf90_def_var(ncid, "lat", NF90_REAL, y_dimid, y_varid))
```

Now we can add units for the coordinate variables

```
CALL check(nf90_put_att(ncid, x_varid, ”units”, “deg”))
CALL check(nf90_put_att(ncid, y_varid, “units”, “deg”))
```

Next we define our data array:

```
dimids = (/ x_dimid, y_dimid /)

!Define variable
CALL check(nf90_def_var(ncid, "Perturbations", NF90_FLOAT, dimids, varid))
```

And now we can add units to the data array:

```
CALL check(nf90_put_att(ncid, varid, "units", "dVs %"))
```

Besides units we can add any other type of attribute we like. Some conventions have named attributes which individual software packages will understand – e.g., the "units" attribute given above is standard. But, we can add anything we like. For example, in this example I may have used a special kind of function, e.g., a Matern function, to create to the perturbations that I want to document in the file. I can add this by:

```
CALL check(nf90_put_att(ncid, varid, "Function_Type", "Matern"))
```

We can also add global attributes such as a title where instead of giving a variable ID we use the *F90_GLOBAL* specifier:

```
CALL check(nf90_put_att(ncid,NF90_GLOBAL, "title", "Realization by KL
Expansion"))
```

Now that I am done adding attributes I must end the definining stage:

```
CALL check(nf90_enddef(ncid)) !End Definitions
```

And now I can write the data:

```
!Write Data
CALL check(nf90_put_var(ncid, x_varid, xpos))
CALL check(nf90_put_var(ncid, y_varid, ypos))
CALL check(nf90_put_var(ncid, varid, idata))
CALL check(nf90_close(ncid))
```

VIII. Example – Irregularly spaced data

There is no real standard for handling irregularly spaced data. However, some conventions have been defined for specific packages. Here I show how to write irregularly spaced data into a format that can be read by OpenDX.

Here is a sample program to generate some test data on an irregular grid:

```
PROGRAM ex
IMPLICIT NONE
REAL(KIND=4), DIMENSION(:,:), ALLOCATABLE :: positions
REAL(KIND=4), DIMENSION(:), ALLOCATABLE :: mydata
REAL(KIND=4) :: x, y
INTEGER(KIND=4) :: NX, NY, naxes
INTEGER(KIND=4) :: J
CHARACTER(LEN=50) :: outfile
```

```
outfile = 'ex1.nc'
NX = 2
NY = 2
naxes = 2

ALLOCATE(mydata(NX*NY))
ALLOCATE(positions(naxes,NX*NY))

positions(1:2,1) = (/0.0, 0.0/)
positions(1:2,2) = (/0.0, 0.2/)
positions(1:2,3) = (/0.3, 0.5/)
positions(1:2,4) = (/0.4, 0.9/)

DO J=1,NX*NY
  x = positions(1,J)
  y = positions(2,J)
  mydata(J) = cos(sqrt(x*x+y*y)/10.0)
ENDDO

!write netCDF file
CALL wirrgrid(outfile,positions,mydata,NX*NY,naxes)

END PROGRAM ex
```

The grid positions are stored in an array, where each line of the array gives the x, and y coordinate (and z, etc. if appropriate) for each data value in the vector *mydata*.

The accompanying subroutine to write the netCDF file is:

```
!WIRRGRID - write a netCDF gridfile
!:=============================================================================
SUBROUTINE wirrgrid(outfile,positions,idata,pointnums,axes)
USE netcdf
IMPLICIT NONE
REAL(KIND=4), DIMENSION(axes,pointnums), INTENT(IN) :: positions
REAL(KIND=4), DIMENSION(pointnums), INTENT(IN) :: idata
INTEGER(KIND=4) :: ncid, x_dimid, y_dimid, d_dimid
INTEGER(KIND=4) :: x_varid, y_varid, varid
INTEGER(KIND=4), DIMENSION(2) :: dimids
INTEGER(KIND=4), INTENT(IN) :: pointnums, axes
INTEGER(KIND=4) :: J, K, c
CHARACTER(LEN=50), INTENT(IN) :: outfile
```

```
!Create the netCDF file.
CALL check(nf90_create(outfile, NF90_CLOBBER, ncid))

!Define the dimensions.
CALL check(nf90_def_dim(ncid, "pointnums", pointnums, x_dimid))
CALL check(nf90_def_dim(ncid, "axes", axes, y_dimid))

!Dimension ID's
dimids = (/ y_dimid, x_dimid /)

!Define coordinate variables
CALL check(nf90_def_var(ncid, "grid", NF90_FLOAT, dimids, x_varid))

!Define data variable
CALL check(nf90_def_var(ncid, "Perturbations", NF90_FLOAT, x_dimid, varid))

!Add attributes
CALL check(nf90_put_att(ncid,varid,"units","%"))
CALL check(nf90_put_att(ncid,varid,"field","Perturbations, vector"))
CALL check(nf90_put_att(ncid,varid,"positions","grid"))

CALL check(nf90_enddef(ncid)) !End Definitions

!Write Data
CALL check(nf90_put_var(ncid, x_varid, positions))
CALL check(nf90_put_var(ncid, varid, idata))
CALL check(nf90_close(ncid))

END SUBROUTINE wirrgrid
!:=============================================================================
```

In this example, using OpenDX, the variable with our data, *Perturbations*, must have the *field* and *positions* attribute set as above so OpenDX knows where to look for the grid positions.

IX. Reading netCDF with Matlab

To read netCDF files into matlab a couple collections of m-files are required. The first is the CSIRO netCDF/OPeNDAP interface to matlab. These are just a collection of m-files which can be obtained from: http://www.marine.csiro.au/sw/matlab-netcdf.html#installation

To use them you should grab them (*cp –r*) and add them to your matlab search path, e.g., in matlab:

```
>> addpath /home/mthorne/applications/matlab/mfiles/matlab_netCDF_OPeNDAP/
```

You also need the *mexnc* netCDF mexfiles. These can be grabbed at http://mexcdf.sourceforge.net/downloads

As above you just need to add these to your search path.

Now once in matlab you can simply read a netCDF file by:

```
>> mydata = getnc('foo.nc');
```

This will interactively prompt you for which variable you want. If e.g., *foo.nc* contains a 2-D grid file written in a matrix, you can return this data into the variable mydata and then start to do work on it, e.g., **`imagesc(mydata)`**.