NCO User's Guide

A suite of netCDF operators

Edition 1.1.47, for NCO Version 1.1.47

May 2000

by Charles S. Zender

Department of Earth System Science

University of California at Irvine

This is the first edition of the *NCO User's Guide*,
and is consistent with version 2 of 'texinfo.tex'.

Published by Charlie Zender
Department of Earth System Science
University of California at Irvine
Irvine, CA 92612-3100 USA

Foreword

NCO is the result of software needs that arose while I worked on projects funded by NCAR, NASA, and ARM. Thinking they might prove useful as tools or templates to others, it is my pleasure to provide them freely to the scientific community. Many users (most of whom I have never met) have encouraged the development of NCO. Thanks espcially to Jan Polcher, Keith Lindsay, Arlindo da Silva, John Sheldon, and William Weibel for stimulating suggestions and correspondence. Your encouragment motivated me to complete the *NCO User's Guide*. So if you like NCO, send me a note! I should mention that NCO is not connected to or officially endorsed by Unidata, ACD, ASP, CGD, or Nike.

Charlie Zender

May 1997
Boulder, Colorado

Summary

This manual describes NCO, which stands for netCDF Operators. NCO is a suite of programs known as *operators*. Each operator is a standalone, command line program which is executed at the UNIX (or NT) shell-level like, e.g., `ls` or `mkdir`. The operators take netCDF file(s) as input, perform an operation (e.g., averaging or hyperslabbing), and produce a netCDF file as output. The operators are primarily designed to aid manipulation and analysis of data. The examples in this documentation are typical applications of the operators for processing climate model output. This reflects their origin, but the operators are as general as netCDF itself.

1 Introduction

1.1 Availability

The complete NCO source distribution is currently distributed as a *compressed tarfile* from `ftp://ftp.cgd.ucar.edu/pub/zender/nco/nco.tar.gz`. The compressed tarfile must be uncompressed and untarred before building NCO. Uncompress the file with '`gunzip nco.tar.gz`'. Extract the source files from the resulting tarfile with '`tar -xvf nco.tar`'. GNU `tar` lets you perform both operations in one step with '`tar -xvzf nco.tar.gz`'.

The documentation for NCO is called the *NCO User's Guide*. The *User's Guide* is available in Postscript, HTML, DVI, TEXinfo, and Info formats. These formats are included in the source distribution in the files '`nco.ps`', '`nco.html`', '`nco.dvi`', '`nco.texi`', and '`nco.info*`', respectively. All the documentation descends from a single source file, '`nco.texi`'[1]. Hence the documentation in every format is very similar. However, some of the complex mathematical expressions needed to describe `ncwa` can only be displayed in the Postscript and DVI formats.

If you want to quickly see what the latest improvements in NCO are (without downloading the entire source distribution), visit the NCO homepage at URL `http://www.cgd.ucar.edu/cms/nco`. The HTML version of the *User's Guide* is also available online through the World Wide Web at URL `http://www.cgd.ucar.edu/cms/nco/nco.html`. To build and use NCO, you must have netCDF installed. The netCDF homepage is `http://www.unidata.ucar.edu/packages/netcdf`.

1.2 Operating systems compatible with NCO

NCO has been successfully ported and tested on the following platforms: GNU/Linux, SunOS 4.1.x, Solaris 2.x, IRIX 5.x and 6.x (including 64-bit architectures), UNICOS 8.x–10.x, AIX 4.x, DEC OSF, and Windows NT4. If you port the code to a new operating system, please send me a note and any patches you required.

The major prerequisite for installing NCO on a particular platform is the successful, prior installation of the netCDF libraries themselves. Unidata

---

[1] To produce these formats, '`nco.texi`' was simply run through the freely available programs `texi2dvi`, `dvips`, `texi2html`, and `makeinfo`. Due to a bug in TEX, the resulting Postscript file, '`nco.ps`', contains the Table of Contents as the final pages. Thus if you print '`nco.ps`', remember to insert the Table of Contents after the cover sheet before you staple the manual.

has shown a commitment to maintaining netCDF on all popular UNIX platforms, and is moving towards full support for Windows NT. Given this, the only difficulty in implementing NCO on a particular platform is standardization of various C and Fortran interface and system calls. The C-code has been tested for ANSI compliance by compiling with GNU `gcc -ansi -pedantic`. Certain branches in the code were required to satisfy the native SGI and SunOS `cc` compilers, which are strictly ANSI compliant and do not allow variable-size arrays, a nice feature supported by GNU, UNICOS, Solaris, and AIX compilers.

The most time-intensive portion of NCO execution is spent in arithmetic operations, e.g., multiplication, averaging, subtraction. Until August, 1999, these operations were performed in Fortran by default. This was a design decision based on the speed of Fortran-based object code vs. C-based object code in late 1994. Since 1994 native C compilers have improved their vectorization capabilities and it has become advantageous to replace all Fortran subroutines with C subroutines. Furthermore, this greatly simplifies the task of compiling on nominally unsupported platforms. As of August 1999, NCO is built entirely in C by default. This allows NCO to compile on any machine with an ANSI C compiler. Furthermore, NCO automatically takes advantage of extensions to ANSI C when compiled with the GNU compiler collection, GNU CC.

It is still possible to request Fortran routines to perform arithmetic operations, however. This can be accomplished by defining the preprocessor token `USE_FORTRAN_ARITHMETIC` and rebuilding NCO. As its name suggests, the `USE_FORTRAN_ARITHMETIC` token instructs NCO to attempts to interface the C routines with Fortran arithmetic. Although using Fortran calls instead of C reduces the portability and and increases the maintenance of the NCO operators, it may also increase the performance of the numeric operators. Presumably this will depend on your machine type, the quality of the C and Fortran compilers, and the size of the data files[2].

1.2.1 Compiling NCO for Windows NT

NCO has been successfully ported and tested on the Microsoft Windows NT 4.0 operating system. The switches necessary to accomplish this are included in the standard distribution of NCO. Using the freely available

[2] If you decide to test the efficiency of the averagers compiled with `USE_FORTRAN_ARITHMETIC` versus the default C averagers I would be most interested to hear the results. Please E-mail me the results including the size of the datasets, the platform, and the change in the wallclock time for execution.

Cygwin (formerly gnu-win32) development environment[3], the compilation process is very similar to installing NCO on a UNIX system. The preprocessor token PVM_ARCH should be set to WIN32. Note that defining WIN32 has the side effect of disabling Internet features of NCO (see below). Unless you have a Fortran compiler (like g77 or f90) available, no other tokens are required. Users with fast Fortran compilers may wish to activate the Fortran arithmetic routines. To do this, define the preprocessor token USE_FORTRAN_ARITHMETIC in the makefile which comes with NCO, 'Makefile', or in the compilation shell.

The least portable section of the code is the use of standard UNIX and Internet protocols (e.g., ftp, rcp, scp, getuid, gethostname, and header files '<arpa/nameser.h>' and '<resolv.h>'). Fortunately, these UNIXy calls are only invoked by the single NCO subroutine which is responsible for retrieving files stored on remote systems (see Section 3.2 [Remote storage], page 19). In order to support NCO on the Windows NT platforms, this single feature was disabled (on Windows NT only). This was required by Cygwin 18.x—newer versions of Cygwin may support these protocols (let me know if this is the case). The NCO operators should behave identically on Windows NT and UNIX platforms in all other respects.

1.3 Libraries

Like all executables, the NCO operators can be built using dynamic linking. This reduces the size of the executable and can result in significant performance enhancements on multiuser systems. Unfortunately, if your library search path (usually the LD_LIBRARY_PATH environment variable) is not set correctly, or if the system libraries have been moved, renamed, or deleted since NCO was installed, it is possible an NCO operator will fail with a message that it cannot find a dynamically loaded (aka *shared object* or '.so') library. This usually produces a distinctive error message, such as 'ld.so.1: /usr/local/bin/ncea: fatal: libsunmath.so.1: can't open file: errno=2'. If you received an error message like this, ask your system administrator to diagnose whether the library is truly missing[4], or whether you simply need to alter your library search path. As a final remedy, you can reinstall NCO with all operators statically linked.

[3] The Cygwin package is available from
http://sourceware.cygnus.com/cygwin
Currently, Cygwin 20.x comes with the GNU C/C++/Fortran compilers (gcc, g++, g77). These GNU compilers may be used to build the netCDF distribution itself.

[4] The ldd command, if it is available on your system, will tell you where the executable is looking for each dynamically loaded library. Use, e.g., ldd `which ncea`.

1.4 netCDF 2.x vs. 3.x

NCO began with netCDF 2.x in 1994. netCDF 3.0 was released in 1996, and we are eager to reap the performance advantages of the newer netCDF implementation. One netCDF 3.x interface call (`nc_inq_libvers`) was added to NCO in January, 1998, to aid in maintainance and debugging. To support this call, NCO must be built with netCDF 3.x releases. Currently, the rest of NCO still uses the netCDF 2.x interface, but, because of the single `nc_inq_libvers` call, NCO no longer builds with netCDF 2.x releases.

However, the ability to compile NCO with only netCDF 2.x calls is worth maintaining because HDF[5] (available from `http://hdf.ncsa.uiuc.edu`) supports only the netCDF 2.x library calls. If NCO is built with only netCDF 2.x calls then some NCO operators will work with HDF files as well as netCDF files[6]. Therefore, the preprocessor token `NETCDF2_ONLY` has been implemented in NCO to eliminate all netCDF 3.x calls. If, at compilation time, `NETCDF2_ONLY` is defined, then NCO will not use any netCDF 3.x calls and the resulting NCO operators should work with HDF files. Note that there are multiple versions of HDF. Currently HDF version 4.x supports netCDF 2.x and thus NCO. HDF version 5.x became available in 1999, but did not support netCDF (or, for that matter, Fortran) as of December 1999. Support for netCDF 3.x in HDF 5.x is being worked on and we will attempt to convert NCO to netCDF 3.x once this is complete.

1.5 Reporting bugs

Please read the manual before sending me any bug reports. Sending me questions whose answers are not in the manual is the best way to motivate me to write more documentation. I would also like to accentuate the contrapositive of this statement. If you think you have found a real bug: simplify the problem to a manageable size, document your run-time environment, and send me the exact error messages (and run the operator with '`-D 5`' to increase the verbosity of the debugging output). Write a bug report based on the latest version of NCO. Send the bug report to `zender@uci.edu`.

---

[5] The Hierarchical Data Format, or HDF, is another self-describing data format similar to, but more elaborate than, netCDF.

[6] I have never tried this but other NCO users have confirmed this is true—it has something to do with linking to the MFHDF library in addition to or instead of the usual netCDF library. Apparently MFHDF only supports netCDF 2.x calls. Thus I will try to keep this capability in NCO as long as it is not too much trouble. If you know which NCO operations should/should not work with HDF files, please let me know.

2 Operator Strategies

2.1 NCO operator philosophy

The main design goal has been to produce operators that can be invoked from the command line to perform useful operations on netCDF files. Many scientists work with models and observations which produce too much data to analyze in tabular format. Thus, it is often natural to reduce and massage this raw or primary level data into summary, or second level data, e.g., temporal or spatial averages. These second level data may become the inputs to graphical and statistical packages, and are often more suitable for archival and dissemination to the scientific community. NCO performs a suite of operations useful in manipulating data from the primary to the second level state. Higher level interpretive languages (e.g., IDL, Yorick, Matlab, NCL, Perl, Python), and lower level compiled languages (e.g., C, Fortran) can always perform any task performed by NCO, but often with more overhead. NCO, on the other hand, is limited to a much smaller set of arithmetic and metadata operations than these full blown languages.

Another goal has been to implement enough command line switches so that frequently used sequences of these operators can be executed from a shell script or batch file. Finally, NCO was written to consume the absolute minimum amount of system memory required to perform a given job. The arithmetic operators are extremely efficient; their exact memory usage is detailed in Section 2.8 [Memory usage], page 14.

2.2 Climate model paradigm

NCO was developed at NCAR to aid analysis and manipulation of datasets produced by General Circulation Models (GCMs). Datasets produced by GCMs share many features with all gridded scientific datasets and so provide a useful paradigm for the explication of the NCO operator set. Examples in this manual use a GCM paradigm because latitude, longitude, time, temperature and other fields related to our natural environment are as easy to visualize for the layman as the expert.

2.3 Temporary output files

NCO operators are designed to be reasonably fault tolerant, so that if there is a system failure or the user aborts the operation (e.g., with `C-c`), then no data is lost. The user-specified *output-file* is only created upon successful completion of the operation[1]. This is accomplished by performing all operations in a temporary copy of *output-file*. The name of the temporary output file is constructed by appending `.pid`<*process ID*>`.`<*operator name*>`.tmp` to the user-specified *output-file* name. When the operator completes its task with no fatal errors, the temporary output file is moved to the user-specified *output-file*. Note the construction of a temporary output file uses more disk space than just overwriting existing files "in place" (because there may be two copies of the same file on disk until the NCO operation successfully concludes and the temporary output file overwrites the existing *output-file*). Also, note this feature increases the execution time of the operator by approximately the time it takes to copy the *output-file*. Finally, note this feature allows the *output-file* to be the same as the *input-file* without any danger of "overlap".

Other safeguards exist to protect the user from inadvertently overwriting data. If the *output-file* specified for a command is a pre-existing file, then the operator will prompt the user whether to overwrite (erase) the existing *output-file*, attempt to append to it, or abort the operation. However, in processing large amounts of data, too many interactive questions can be a curse to productivity. Therefore NCO also implements two ways to override its own safety features, the '`-O`' and '`-A`' switches. Specifying '`-O`' tells the operator to overwrite any existing *output-file* without prompting the user interactively. Specifying '`-A`' tells the operator to attempt to append to any existing *output-file* without prompting the user interactively. These switches are useful in batch environments because they suppress interactive keyboard input.

2.4 Appending variables to a file

A frequently useful operation is adding variables from one file to another. This is referred to as *appending*, although some prefer the terminology *merging*[2] or *pasting*. Appending is often confused with what NCO calls *concatenation*. In NCO, concatenation refers to splicing a variable along the record dimension. Appending, on the other hand, refers to adding variables from one file to another[3]. In this sense, `ncks` can append variables from one file to another file. This capability is invoked by naming two files on the command line, *input-file* and *output-file*. When *output-file* already exists, the user is

---

[1] The `ncrename` operator is an exception to this rule. See Section 4.9 [ncrename netCDF Renamer], page 55.

[2] The terminology *merging* is reserved for an (unwritten) operator which replaces hyperslabs of a variable in one file with hyperslabs of the same variable from another file

[3] Yes, the terminology is confusing. By all means mail me if you think of a better nomenclature. Should NCO use *paste* instead of *append*?

prompted whether to *overwrite*, *append/replace*, or *exit* from the command. Selecting *overwrite* tells the operator to erase the existing *output-file* and replace it with the results of the operation. Selecting *exit* causes the operator to exit—the *output-file* will not be touched in this case. Selecting *append/replace* causes the operator to attempt to place the results of the operation in the existing *output-file*, See Section 4.6 [ncks netCDF Kitchen Sink], page 45.

2.5 Averagers vs. Concatenators

The most frequently used operators of NCO are probably the averagers and concatenators. Because there are so many permutations of averaging (e.g., across files, within a file, over the record dimension, over other dimensions, with or without weights and masks) and of concatenating (across files, along the record dimension, along other dimensions), there are currently no fewer than five operators which tackle these two purposes: `ncra`, `ncea`, `ncwa`, `ncrcat`, and `ncecat`. These operators do share many capabilities[4], but each has its unique specialty. Two of these operators, `ncrcat` and `ncecat`, are for concatenating hyperslabs across files. The other two operators, `ncra` and `ncea`, are for averaging hyperslabs across files[5]. First, let's describe the concatenators, then the averagers.

2.5.1 Concatenators `ncrcat` and `ncecat`

Joining independent files together along a record coordinate is called *concatenation*. `ncrcat` is designed for concatenating record variables, while `ncecat` is designed for concatenating fixed length variables. Consider 5 files, '`85.nc`', '`86.nc`', ... '`89.nc`' each containing a year's worth of data. Say you wish to create from them a single file, '`8589.nc`' containing all the data, i.e., spanning all 5 years. If the annual files make use of the same record variable, then `ncrcat` will do the job nicely with, e.g., `ncrcat 8?.nc 8589.nc`. The number of records in the input files is arbitrary and can vary from file to file. See Section 4.8 [ncrcat netCDF Record Concatenator], page 53, for a complete description of `ncrcat`.

[4] Currently `ncea` and `ncrcat` are symbolically linked to the `ncra` executable, which behaves slightly differently based on its invocation name (i.e., '`argv[0]`'). These three operators share the same source code, but merely have different inner loops.

[5] The third averaging operator, `ncwa`, is the most sophisticated averager in NCO. However, `ncwa` is in a different class than `ncra` and `ncea` because it can only operate on a single file per invocation (as opposed to multiple files). On that single file, however, `ncwa` provides a richer set of averaging options—including weighting, masking, and broadcasting.

However, suppose the annual files have no record variable, and thus their data is all fixed length. For example, the files may not be conceptually sequential, but rather members of the same group, or *ensemble*. Members of an ensemble may have no reason to contain a record dimension. `ncecat` will create a new record dimension (named *record* by default) with which to glue together the individual files into the single ensemble file. If `ncecat` is used on files which contain an existing record dimension, that record dimension will be converted into a fixed length dimension of the same name and a new record dimension will be created. Consider five realizations, '`85a.nc`', '`85b.nc`', ... '`85e.nc`' of 1985 predictions from the same climate model. Then `ncecat 85?.nc 85_ens.nc` glues the individual realizations together into the single file, '`85_ens.nc`'. If an input variable was dimensioned [`lat`,`lon`], it will have dimensions [`record`,`lat`,`lon`] in the output file. A restriction of `ncecat` is that the hyperslabs of the processed variables must be the same from file to file. Normally this means all the input files are the same size, and contain data on different realizations of the same variables. See Section 4.4 [ncecat netCDF Ensemble Concatenator], page 42, for a complete description of `ncecat`.

Note that `ncrcat` cannot concatenate fixed-length variables, whereas `ncecat` can concatenate both fixed-length and record variables. To conserve system memory, use `ncrcat` rather than `ncecat` when concatenating record variables.

2.5.2 Averagers `ncea`, `ncra`, and `ncwa`

The differences between the averagers `ncra` and `ncea` are analogous to the differences between the concatenators. `ncra` is designed for averaging record variables from at least one file, while `ncea` is designed for averaging fixed length variables from multiple files. `ncra` performs a simple arithmetic average over the record dimension of all the input files, with each record having an equal weight in the average. `ncea` performs a simple arithmetic average of all the input files, with each file having an equal weight in the average. Note that `ncra` cannot average fixed-length variables, but `ncea` can average both fixed-length and record variables. To conserve system memory, use `ncra` rather than `ncea` where possible (e.g., if each *input-file* is one record long). The file output from `ncea` will have the same dimensions (meaning dimension names as well as sizes) as the input hyperslabs (see Section 4.3 [ncea netCDF Ensemble Averager], page 40, for a complete description of `ncea`). The file output from `ncra` will have the same dimensions as the input hyperslabs except for the record dimension, which will have a size of 1 (see Section 4.7 [ncra netCDF Record Averager], page 51, for a complete description of `ncra`).

2.6 Working with large numbers of input files

Occasionally one desires to digest (i.e., concatenate or average) hundreds or thousands of input files. One brave user, for example, recently created a five year timeseries of satellite observations by using `ncecat` to join thousands of daily data files together. Unfotunately, data archives (e.g., NASA EOSDIS) are unlikely to distribute netCDF files conveniently named in a format the '`-n` *loop*' switch (which automatically generates arbitrary numbers of input filenames) understands. If there is not a simple, arithmetic pattern to the input filenames (e.g., '`h00001.nc`', '`h00002.nc`', ... '`h90210.nc`') then the '`-n` *loop*' switch is useless. Moreover, when the input files are so numerous that the input filenames are too lengthy (when strung together as a single argument) to be passed by the calling shell to the NCO operator[6], then the following strategy has proven useful to specify the input filenames to NCO. Write a script that creates symbolic links between the irregular input filenames and a set of regular, arithmetic filenames that '`-n` *loop*' switch understands. The NCO operator will then succeed at automatically generating the filnames with the '`-n` *loop*' option (which circumvents any OS and shell limits on command line size). You can remove the symbolic links once the operator completes its task.

2.7 Working with large files

Large files are those files that are comparable in size to the amount of memory (RAM) in your computer. Many users of NCO work with files larger than 100 Mb. Files this large not only push the current edge of storage technology, they present special problems for programs which attempt to access the entire file at once, such as `ncea`, and `ncecat`. If you need to work with a 300 Mb file on a machine with only 32 Mb of memory then you will need large amounts of swap space (virtual memory on disk) and NCO will work slowly, or else NCO will fail. There is no easy solution for this and the best strategy is to work on a machine with massive amounts of memory and swap space. That is, if your local machine has problems working with large files, try running NCO from a more powerful machine, such as a network server. Certain machine architectures, e.g., Cray UNICOS, have special commands which allow one to increase the amount of interactive memory. If

[6] The exact length which exceeds the operating system internal limit for command line lengths varies from OS to OS and from shell to shell. GNU `bash` may not have any arbitrary fixed limits to the size of command line arguments. Many OSs cannot handle command line arguments longer than a few thousand characters. When this occurs, the ANSI C-standard `argc-argv` method of passing arguments from the calling shell to a C-program (i.e., an NCO operator) breaks down.

you get a core dump on a Cray system (e.g., '`Error exit (core dumped)`'), try increasing the available memory by using the `ilimit` command.

The speed of the NCO operators also depends on file size. When processing large files the operators may appear to hang, or do nothing, for large periods of time. In order to see what the operator is actually doing, it is useful to activate a more verbose output mode. This is accomplished by supplying a number greater than 0 to the '`-D` *debug_level*' switch. When the *debug_level* is nonzero, the operators report their current status to the terminal through the *stderr* facility. Using '`-D`' does not slow the operators down. Choose a *debug_level* between 1 and 3 for most situations, e.g., `ncea -D 2 85.nc 86.nc 8586.nc`. A full description of how to estimate the actual amount of memory the multi-file NCO operators consume is given in Section 2.8 [Memory usage], page 14.

2.8 Approximate NCO memory requirements

The multi-file operators currently comprise the record operators, `ncra` and `ncrcat`, and the ensemble operators, `ncea` and `ncecat`, The record operators require *much less* memory than the ensemble operators. This is because the record operators are designed to operate on a single record of a file at a time, while the ensemble operators must retrieve an entire variable at a time into memory. Let MS be the peak sustained memory demand of an operator, FT be the memory required to store the entire contents of the variables to be processed in an input file, FR be the memory required to store the entire contents of a single record of the variables to be processed in an input file, VR be the memory required to store a single record of the largest record variable to be processed in an input file, VT be the memory required to store the largest variable to be processed in an input file, VI be the memory required to store the largest variable which is not processed, but is copied from the initial file to the output file. All operators require $MI = VI$ during the initial copying of variables from the first input file to the output file. This is the *initial* (and transient) memory demand. The *sustained* memory demand is that memory required by the operators during the processing (i.e., averaging, concatenation) phase which lasts until all the input files have been processed. The operators have the following memory requirements: `ncrcat` requires $MS <= VR$. `ncecat` requires $MS <= VT$. `ncra` requires $MS = 2FR + VR$. `ncea` requires $MS = 2FT + VT$. Note that only variables which are processed, i.e., averaged or concatenated, contribute to MS. Memory is never allocated to hold variables which do not appear in the output file (see Section 3.4 [Variable subsetting], page 21).

2.9 Performance limitations of the operators

1. No buffering of data is performed during `ncvarget` and `ncvarput` operations. Hyperslabs too large too hold in core memory will suffer substantial performance penalties because of this.
2. Since coordinate variables are assumed to be monotonic, the search for bracketing the user-specified limits should employ a quicker algorithm, like bisection, than the two-sided incremental search currently implemented.
3. *C_format*, *FORTRAN_format*, *signedness*, *scale_format* and *add_offset* attributes are ignored by `ncks` when printing variables to screen.
4. Some random access operations on large files on certain architectures (e.g., 400 Mb on UNICOS) are *much* slower with these operators than with similar operations performed using languages that bypass the netCDF interface (e.g., Yorick). The cause for this is not understood at present.

3 Features common to most operators

Many features have been implemented in more than one operator and are described here for brevity. The description of each feature is preceded by a box listing the operators for which the feature is implemented. Command line switches for a given feature are consistent across all operators wherever possible. If no "key switches" are listed for a feature, then that particular feature is automatic and cannot be controlled by the user.

3.1 Specifying input files

Availability: All operators
Key switches: '-n', '-p'

It is important that the user be able to specify multiple input files without tediously typing in each by its full name. There are four different ways of specifying input files to NCO: explicitly typing each, using UNIX shell wildcards, and using the NCO '-n' and '-p' switches. To illustrate these methods, consider the simple problem of using `ncra` to average five input files, '85.nc', '86.nc', ... '89.nc', and store the results in '8589.nc'. Here are the four methods in order. They produce identical answers.

```
ncra 85.nc 86.nc 87.nc 88.nc 89.nc 8589.nc
ncra 8[56789].nc 8589.nc
ncra -p input-path 85.nc 86.nc 87.nc 88.nc 89.nc 8589.nc
ncra -n 5,2,1 85.nc 8589.nc
```

The first method (explicitly specifying all filenames) works by brute force. The second method relies on the operating system shell to *glob* (expand) the *regular expression* `8[56789].nc`. The shell passes valid filenames which match the expansion to `ncra`. The third method uses the '-p *input-path*' argument to specify the directory where all the input files reside. NCO prepends *input-path* (e.g., '/data/usrname/model') to all *input-files* (but not to *output-file*). Thus, using '-p', the path to any number of input files need only be specified once. Note *input-path* need not end with '/'; the '/' is automatically generated if necessary.

The last method passes (with '-n') syntax concisely describing the entire set of filenames[1]. This option is only available with the *multi-file operators*: `ncra`, `ncrcat`, `ncea`, and `ncecat`. By definition, multi-file operators are able to process an arbitrary number of *input-files*. This

[1] The '-n' option is a backward compatible superset of the `NINTAP` option from the NCAR CCM Processor.

option is very useful for abbreviating lists of filenames representable as *alphanumeric_prefix*+*numeric_suffix*+'.'+*filetype* where *alphanumeric_prefix* is a string of arbitrary length and composition, *numeric_suffix* is a fixed width field of digits, and *filetype* is a standard filetype indicator. For example, in the file 'ccm3_h0001.nc', we have *alphanumeric_prefix* = 'ccm3_h', *numeric_suffix* = '0001', and *filetype* = 'nc'.

NCO is able to decode lists of such filenames encoded using the '-n' option. The simpler (3-argument) '-n' usage takes the form -n *file˙number*,*digit˙number*,*numeric˙increment* where *file_number* is the number of files, *digit_number* is the fixed number of numeric digits comprising the *numeric_suffix*, and *numeric_increment* is the constant, integer-valued difference between the *numeric_suffix* of any two consecutive files. The value of *alphanumeric_prefix* is taken from the input file, which serves as a template for decoding the filenames. In the example above, the encoding -n 5,2,1 along with the input file name '85.nc' tells NCO to construct five (5) filenames identical to the template '85.nc' except that the final two (2) digits are a numeric suffix to be incremented by one (1) for each successive file. Currently *filetype* may be either be empty, 'nc', 'cdf', 'hdf', or 'hd5'. If present, these *filetype* suffixes (and the preceding '.') are ignored by NCO as it uses the '-n' arguments to locate, evaluate, and compute the *numeric_suffix* component of filenames.

Recently the '-n' option has been extended to allow convenient specification of filenames with "circular" characteristics. This means it is now possible for NCO to automatically generate filenames which increment regularly until a specified maximum value, and then wrap back to begin again at a specified minimum value. The corresponding '-n' usage becomes more complex, taking one or two additional arguments for a total of four or five, respectively: -n *file˙number*,*digit˙number*,*numeric˙increment*[,*numeric˙max*[,*numeric˙min*]] where *numeric_max*, if present, is the maximum integer-value of *numeric_suffix* and *numeric_min*, if present, is the minimum integer-value of *numeric_suffix*. Consider, for example, the problem of specifying non-consecutive input files where the filename suffixes end with the month index. In climate modeling it is common to create summertime and wintertime averages which contain the averages of the months June–July–August, and December–January–February, respectively:

```
ncra -n 3,2,1 85_06.nc 85_0608.nc
ncra -n 3,2,1,12 85_12.nc 85_1202.nc
ncra -n 3,2,1,12,1 85_12.nc 85_1202.nc
```

The first example shows that three arguments to the '-n' option suffice to specify consecutive months (06, 07, 08) which do not "wrap" back to a minimum value. The second example shows how to use the optional fourth and fifth elements of the '-n' option to specify a wrap value to NCO. The fourth argument to '-n', if present, specifies the maximum integer value of *numeric_suffix*. In this case the maximum value is 12, and will be formatted as '12' in the filename string. The fifth argument to '-n', if present, specifies the minimum integer value of *numeric_suffix*. The default minimum filename suffix is 1, which is formatted as '01' in this case. Thus the second and third examples have the same effect, that is, they automatically generate, in order, the filenames '85_12.nc', '85_01.nc', and '85_02.nc' as input to NCO.

3.2 Accessing files stored remotely

Availability: All operators
Key switches: '-p', '-l', '-R'

All NCO operators can retrieve files from remote sites as well as the local file system. A remote site can be an anonymous FTP server, a machine on which the user has `rcp(1)` or `scp(1)` privileges, or NCAR's Mass Storage System (MSS). To access a file via an anonymous FTP server, supply the remote file's URL. To access a file using `rcp(1)` or `scp(1)`, specify the Internet address of the remote file. Of course in this case you must have `rcp(1)` or `scp(1)` privileges which allow transparent (no password entry required) access to the remote machine. This means that '~/.rhosts' or '~/ssh/authorized_keys' must be set accordingly on both local and remote machines.

To access a file on NCAR's MSS, specify the full MSS pathname of the remote file. NCO will attempt to detect whether the local machine has direct (synchronous) MSS access. In this case, NCO attempts to use the NCAR `msrcp` command[2], or, failing that, `/usr/local/bin/msread`. Otherwise NCO attempts to retrieve the MSS file through the (asynchronous) Masnet Interface Gateway System (MIGS) using the `nrnet` command.

The following examples show how one might analyzed files stored on remote systems.

```
ncks -H -l ./ ftp://ftp.cgd.ucar.edu/pub/zender/nc/in.nc
ncks -H -l ./ dust.ps.uci.edu:/home/zender/nc/in.nc
ncks -H -l ./ /ZENDER/nc/in.nc
```

The first example should work verbatim on your system if your system is connected to the Internet and is not behind a firewall. The second example should fail on your system unless you have `rcp` or `scp` priveleges on the

[2] The `msrcp` command must be in the user's path and located in one of the following directories: `/usr/local/bin`, `/usr/bin`, `/opt/local/bin`, or `/usr/local/dcs/bin`.

machine `dust.ps.uci.edu`. The third example should work from NCAR computers with local access to the `msrcp`, `msread`, or `nrnet` commands. The above commands can be rewritten using the '`-p` *input-path*' option as follows:

```
ncks -H -p ftp://ftp.cgd.ucar.edu/pub/zender/nc -l ./ in.nc
ncks -H -p dust.ps.uci.edu:/home/zender/nc -l ./ in.nc
ncks -H -p /ZENDER/nc -l ./ in.nc
```

Using '`-p`' is recommended because it clearly separates the *input-path* from the filename itself, sometimes called the *stub*. When *input-path* is not explicitly specified using '`-p`', NCO internally generates an *input-path* from the first input filename. The automatically generated *input-path* is constructed by stripping the input filename of everything following the final '`/`' character (i.e., removing the stub). The '`-l` *output-path*' option tells NCO where to store the remotely retrieved file and the output file. Often the path to a remotely retrieved file is quite different than the path on the local machine where you would like to store the file. If '`-l`' is not specified then NCO internally generates an *output-path* by simply setting *output-path* equal to *input-path* stripped of any machine names. If '`-l`' is not specified and the remote file resides on the NCAR MSS system, then the leading character of *input-path*, '`/`', is also stripped from *output-path*. Specifying *output-path* as '`-l ./`' tells NCO to store the remotely retrieved file and the output file in the current directory. Note that '`-l .`' is equivalent to '`-l ./`' though the latter is syntactically more clear.

Note that this remote retrieval feature can easily be used to retrieve *any* file, including non-netCDF files, via anonymous FTP. Often this method is quicker than using a browser, or running an FTP session from a shell window yourself. For example, say you want to obtain a JPEG file from a weather server.

```
ncks -p ftp://weather.edu/pub/pix/jpeg -l ./ storm.jpg
```

In this example, `ncks` automatically performs an anonymous FTP login to the remote machine and retrieves the specified file. When `ncks` attempts to read the local copy of '`storm.nc`' as a netCDF file, it fails and exits, leaving '`storm.nc`' in the current directory.

3.3 Retention of remotely retrieved files

Availability: All operators
Key switches: '`-R`'

In order to conserve local file system space, files retrieved from remote locations are automatically deleted from the local file system once they have been processed. Many NCO operators were constructed to work with numerous large (e.g., 200 Mb) files. Retrieval of multiple files from remote locations is done serially. Each file is retrieved, processed, then deleted before the cycle repeats. In cases where it is useful to keep the remotely-retrieved files on the local file system after processing, the automatic removal feature may be disabled by specifying '`-R`' on the command line.

3.4 Including/Excluding specific variables

Availability: `ncdiff`, `ncea`, `ncecat`, `ncflint`, `ncks`, `ncra`, `ncrcat`, `ncwa`
Key switches: '`-v`', '`-x`'

Variable subsetting is implemented with the '`-v` *var*`[,...]`' and '`-x`' options. A list of variables to extract is specified following the '`-v`' option, e.g., '`-v time,lat,lon`'. Not using the '`-v`' option is equivalent to specifying all variables. The '`-x`' option causes the list of variables specified with '`-v`' to be excluded rather than extracted. Thus '`-x`' saves typing when you only want to extract fewer than half of the variables in a file. Remember, if you are stretching the limits of your system's memory by averaging or concatenating large files, then the easiest solution is often to use the '`-v`' option to retain only the variables you really need (see Section 2.8 [Memory usage], page 14).

3.5 Including/Excluding coordinate variables

Availability: `ncdiff`, `ncea`, `ncecat`, `ncflint`, `ncks`, `ncra`, `ncrcat`, `ncwa`
Key switches: '`-C`', '`-c`'

By default, coordinates variables associated with any variable appearing in the *output-file* will also appear in the *output-file*, even if they are not explicitly specified, e.g., with the '`-v`' switch. Thus variables with a latitude coordinate `lat` always carry the values of `lat` with them into the *output-file*. This feature can be disabled with '`-C`', which causes NCO to not automatically add coordinates to the variables appearing in the *output-file*. However, using '`-C`' does not preclude the user from including some coordinates in the output files simply by explicitly selecting the coordinates with the *-v* option. The '`-c`' option, on the other hand, is a shorthand way of automatically specifying that *all* coordinate variables in the *input-files* should appear in the *output-file*. Thus '`-c`' allows the user to select all the coordinate variables without having to know their names.

3.6 C & Fortran index conventions

Availability: `ncdiff`, `ncea`, `ncecat`, `ncflint`, `ncks`, `ncra`, `ncrcat`, `ncwa`
Key switches: '`-F`'

By default, NCO uses C-style (0-based) indices for all I/O. The '`-F`' switch tells NCO to switch to reading and writing with Fortran index conventions. In Fortran, indices begin counting from 1 (rather than 0), and dimensions are ordered from fastest varying to slowest varying. Consider a file '`85.nc`' containing 12 months of data in the record dimension `time`. The following hyperslab operations produce identical results, a June-July-August average of the data:

```
ncra -d time,5,7 85.nc 85_JJA.nc
ncra -F -d time,6,8 85.nc 85_JJA.nc
```

Printing variable *three_dmn_var* in file '`in.nc`' first with C indexing conventions, then with Fortran indexing conventions results in the following output formats:

```
% ncks -H -v three_dmn_var in.nc
% lat[0]=-90 lev[0]=1000 lon[0]=-180 three_dmn_var[0]=0
...
% ncks -F -H -v three_dmn_var in.nc
% lon(1)=-180 lev(1)=1000 lat(1)=-90 three_dmn_var(1)=0
```

3.7 Hyperslabs

Availability: `ncdiff`, `ncea`, `ncecat`, `ncflint`, `ncks`, `ncra`, `ncrcat`, `ncwa`
Key switches: '`-d`'

A *hyperslab* is a subset of a variable's data. The coordinates of a hyperslab are specified with the `-d` *dim*`,[`*min*`][,[`*max*`]]` option. The bounds of the hyperslab to be extracted are specified by the associated *min* and *max* values. A half-open range is specified by omitting either the *min* or *max* parameter but including the separating comma. The unspecified limit is interpreted as the maximum or minimum value in the unspecified direction. A cross-section at a specific coordinate is extracted by specifying only the *min* limit and omitting a trailing comma. Dimensions not mentioned are passed with no reduction in range. The dimensionality of variables is not reduced (in the case of a cross-section, the size of the constant dimension will be one). If values of a coordinate-variable are used to specify a range or cross-section, then the coordinate variable must be monotonic (values either increasing or decreasing). In this case, command-line values need not exactly match coordinate values for the specified dimension. Ranges are determined by seeking the first coordinate value to occur in the closed range [*min*,*max*] and including all subsequent values until one falls outside the range. The coordinate value for a cross-section is the coordinate-variable value closest to the specified value and must lie within the range or coordinate-variable values.

Coordinate values should be specified using real notation with a decimal point required in the value, whereas dimension indices are specified using integer notation without a decimal point. Note that this convention is only to differentiate coordinate values from dimension indices, and is independent of the actual type of netCDF coordinate variables, if any. For a given dimension, the specified limits must both be coordinate values (with decimal points) or dimension indices (no decimal points).

User-specified coordinate limits are promoted to double precision values while searching for the indices which bracket the range. Thus, hyperslabs on coordinates of type `NC_BYTE` and `NC_CHAR` are computed numerically rather than lexically, so the results are unpredictable.

The relative magnitude of *min* and *max* indicate to the operator whether to expect a *wrapped coordinate* (see Section 3.8 [Wrapped coordinates], page 24), such as longitude. If *min*>*max*, the NCO expects the coordinate to be wrapped, and a warning message will be printed. When this occurs, NCO selects all values outside the domain [*max*<*min*], i.e., all the values exclusive of the values which would have been selected if *min* and *max* were swapped. If this seems confusing, test your command on just the coordinate variables with `ncks`, and then examine the output to ensure NCO selected the hyperslab you expected (coordinate wrapping is only supported by `ncks`).

Because of the way wrapped coordinates are interpreted, it is very important to make sure you always specify hyperslabs in the monotonically increasing sense, i.e., *min*<*max* (even if the underlying coordinate variable is monotonically decreasing). The only exception to this is when you are indeed specifying a wrapped coordinate. The distinction is crucial to understand because the points selected by, e.g., `-d longitude,50.,340.`, are exactly the complement of the points selected by `-d longitude,340.,50.`.

Not specifying any hyperslab option is equivalent to specifying full ranges of all dimensions. This option may be specified more than once in a single command (each hyperslabed dimension requires its own `-d` option).

3.8 Wrapped coordinates

Availability: `ncks`
Key switches: '`-d`'

A *wrapped coordinate* is a coordinate whose values increase or decrease monotonically (nothing unusual so far), but which represents a dimension that ends where it begins (i.e., wraps around on itself). Longitude (i.e., degrees on a circle) is a familiar example of a wrapped coordinate. Longitude increases to the East of Greenwich, England, where it is defined to be zero. Halfway around the globe, the longitude is 180 degrees East (or West). Continuing eastward, longitude increases to 360 degrees East at Greenwich. The longitude values of most geophysical data is either in the range [0,360), or [-180,180). In either case, the Westernmost and Easternmost longitudes are numerically separated by 360 degrees, but represent contiguous regions on the globe. For example, the Saharan desert stretches from roughly 340 to 50 degrees East. Extracting the hyperslab of data representing the Sahara from a global dataset presents special problems when the global dataset is stored consecutively in longitude from 0 to 360 degrees. This is because the data for the Sahara will not be contiguous in the *input-file* but is expected by the user to be contiguous in the *output-file*. In this case, `ncks` must invoke special software routines to assemble the desired output hyperslab from multiple reads of the *input-file*.

Assume the domain of the monotonically increasing longitude coordinate `lon` is $0 < lon < 360$. `ncks` will extract a hyperslab which crosses the Greenwich meridian simply by specifying the westernmost longitude as *min* and the easternmost longitude as *max*. Thus, the following commands extract a hyperslab containing the Saharan desert:

```
ncks -d lon,340.,50. in.nc out.nc
ncks -d lon,340.,50. -d lat,10.,35. in.nc out.nc
```

The first example selects data in the same longitude range as the Sahara. The second example further constrains the data to having the same latitude as the Sahara. The coordinate `lon` in the *output-file*, '`out.nc`', will no longer be monotonic! The values of `lon` will be, e.g., '`340, 350, 0, 10, 20, 30, 40, 50`'. This can have serious implications should you run '`out.nc`' through another operation which expects the `lon` coordinate to be monotonically increasing. Fortunately, the chances of this happening are slim, since `lon` has already been hyperslabbed, there should be no reason to hyperslab `lon` again. Should you need to hyperslab `lon` again, be sure to give dimensional indices as the hyperslab arguments, rather than coordinate values (see Section 3.7 [Hyperslabs], page 22).

3.9 Stride

Availability: `ncks`, `ncra`, `ncrcat`
Key switches: '`-d`'

`ncks` offers support for specifying a *stride* for any hyperslab, while `ncra` and `ncrcat` suport the *stride* argument only for the record dimension. The *stride* is the spacing between consecutive points in a hyperslab. A *stride* of 1 means pick all the elements of the hyperslab, but a *stride* of 2 means skip every other element, etc. Using the *stride* option with `ncra` and `ncrcat` makes it possible, for instance, to average or concatenate regular intervals across multi-file input data sets.

The *stride* is specified as the optional fourth argument to the '`-d`' hyperslab specification: `-d` *dim*`,[`*min*`][,[`*max*`]][,[`*stride*`]]`. Specify *stride* as an integer (i.e., no decimal point) following the third comma in the '`-d`' argument. There is no default value for *stride*. Thus using '`-d time,,,2`' is valid but '`-d time,,,2.0`' and '`-d time,,,`' are not. When *stride* is specified but *min* is not, there is an ambiguity as to whether the extracted hyperslab should begin with (using C-style, 0-based indexes) element 0 or element '`stride-1`'. NCO must resolve this ambiguity and it chooses element 0 as the first element of the hyperslab when *min* is not specified. Thus '`-d time,,,`*stride*' is syntactically equivalent to '`-d time,0,,`*stride*'. This means, for example, that specifying the operation '`-d time,,,2`' on the array '`1,2,3,4,5`' selects the hyperslab '`1,3,5`'. To obtain the hyperslab '`2,4`' instead, simply explicitly specify the starting index as 1, i.e., '`-d time,1,,2`'.

For example, consider a file '`8501_8912.nc`' which contains 60 consecutive months of data. Say you wish to obtain just the March data from this file. Using 0-based subscripts (see Section 3.6 [Fortran indexing], page 22) these data are stored in records 2, 14, ... 50 so the desired *stride* is 12. Without the *stride* option, the procedure is very awkward. One could use `ncks` five times and then use `ncrcat` to concatenate the resulting files together:

```
foreach idx (02 14 26 38 50)
ncks -d time,$idx 8501_8912.nc foo.$idx
end
ncrcat foo.?? 8589_03.nc
rm foo.??
```

With the *stride* option, `ncks` performs this hyperslab extraction in one operation:

```
ncks -d time,2,,12 8501_8912.nc 8589_03.nc
```

See Section 4.6 [ncks netCDF Kitchen Sink], page 45, for more information on `ncks`.

The *stride* option is supported by `ncra` and `ncrcat` for the record dimension only. This makes it possible, for instance, to average or concatenate regular intervals across multi-file input data sets.

```
ncra -F -d time,3,,12 85.nc 86.nc 87.nc 88.nc 89.nc 8589_03.nc
ncrcat -F -d time,3,,12 85.nc 86.nc 87.nc 88.nc 89.nc 8503_8903.nc
```

3.10 Missing values

Availability: `ncdiff`, `ncea`, `ncflint`, `ncra`, `ncwa`
Key switches: None

The phrase *missing data* refers to data points that are missing, invalid, or for any reason not intended to be arithmetically processed in the same fashion as valid data. The NCO arithmetic operators attempt to handle missing data in an intelligent fashion. There are four steps in the NCO treatment of missing data:

1. Identifying variables which may contain missing data.

   NCO follows the convention that missing data should be stored with the *missing_value* specified in the variable's `missing_value` attribute. The *only* way NCO recognizes that a variable *may* contain missing data is if the variable has a `missing_value` attribute. In this case, any elements of the variable which are numerically equal to the *missing_value* are treated as missing data.
2. Converting the *missing_value* to the type of the variable, if neccessary.

   Consider a variable *var* of type *var_type* with a `missing_value` attribute of type *att_type* containing the value *missing_value*. As a guideline, the type of the `missing_value` attribute should be the same as the type of the variable it is attached to. If *var_type* equals *att_type* then NCO straightforwardly compares each value of *var* to *missing_value* to determine which elements of *var* are to be treated as missing data. If not, then NCO will internally convert *att_type* to *var_type* by using the implicit conversion rules of C, or, if *att_type* is `NC_CHAR`[3], by typecasting the results of the C function `strtod(`*missing˙value*`)`. You may use the NCO operator `ncatted` to change the `missing_value` attribute and all data whose data is *missing_value* to a new value (see Section 4.1 [ncatted netCDF Attribute Editor], page 32).
3. Identifying missing data during arithmetic operations.

   When an NCO arithmetic operator is processing a variable *var* with a `missing_value` attribute, it compares each value of *var* to *missing_value*

[3] For example, the DOE ARM program often uses *att_type* = `NC_CHAR` and *missing_value* = '`-99999.`'.

before performing an operation. Note the *missing_value* comparison inflicts a performance penalty on the operator. Arithmetic processing of variables which contain the `missing_value` attribute always incurs this penalty, even when none of the data is missing. Conversely, arithmetic processing of variables which do not contain the `missing_value` attribute never incurs this penalty. In other words, do not attach a `missing_value` attribute to a variable which does not contain missing data. This exhortation can usually be obeyed for model generated data, but it may be harder to know in advance whether all observational data will be valid or not.

4. Treatment of any data identified as missing in arithmetic operators.

   NCO averagers (`ncra`, `ncea`, `ncwa`) do not count any element with the value *missing_value* towards the average. `ncdiff` and `ncflint` define a *missing_value* result when either of the input values is a *missing_value*. Sometimes the *missing_value* may change from file to file in a multi-file operator, e.g., `ncra`. NCO is written to account for this (it always compares a variable to the *missing_value* assigned to that variable in the current file). Suffice it to say that, in all known cases, NCO does "the right thing".

3.11 Suppressing interactive prompts

Availability: All operators
Key switches: '`-O`', '`-A`'

If the *output-file* specified for a command is a pre-existing file, then the operator will prompt the user whether to overwrite (erase) the existing *output-file*, attempt to append to it, or abort the operation. However, in processing large amounts of data, too many interactive questions can be a curse to productivity. Therefore NCO also implements two ways to override its own safety features, the '`-O`' and '`-A`' switches. Specifying '`-O`' tells the operator to overwrite any existing *output-file* without prompting the user interactively. Specifying '`-A`' tells the operator to attempt to append to any existing *output-file* without prompting the user interactively. These switches are useful in batch environments because they suppress interactive keyboard input.

3.12 Operator version

Availability: All operators
Key switches: '`-r`'

All operators can be told to print their internal version number and copyright notice and then quit with the '`-r`' switch. The internal version number varies between operators, and indicates the most recent change to a particular operator's source code. This is useful in making sure you are working with the most recent operators. The version of NCO you are using might be, e.g., 1.0. However using '`-r`' on, say, `ncks`, will produce something like '`ncks 3.24 (1997/05/06) Copyright 1995--1997 University Corporation for Atmospheric Research`'. This tells you `ncks` contains all patches up to version 3.24, which dates from May 6, 1997.

3.13 History attribute

Availability: All operators
Key switches: '`-h`'

All operators automatically append a `history` global attribute to any file they modify or create. The `history` attribute consists of a timestamp and the full string of the invocation command to the operator, e.g., '`Mon May 26 20:10:24 1997: ncks in.nc foo.nc`'. The full contents of an existing `history` attribute are copied from the first *input-file* to the *output-file*. The timestamps appear in reverse chronological order, with the most recent timestamp appearing first in the `history` attribute. Since NCO and many other netCDF operators adhere to the `history` convention, the entire data processing path of a given netCDF file may often be deduced from examination of its `history` attribute. To avoid information overkill, all operators have an optional switch ('`-h`') to override automatically appending the `history` attribute (see Section 4.1 [ncatted netCDF Attribute Editor], page 32).

3.14 NCAR CSM Conventions

Availability: `ncdiff`, `ncea`, `ncecat`, `ncflint`, `ncra`, `ncwa`
Key switches: None

NCO has been programmed to recognize NCAR CSM history tapes. If you do not work with NCAR CSM data then you may skip this section. The CSM netCDF convention is described at `http://www.cgd.ucar.edu/csm/experiments/output.format.html`. Most of the CSM netCDF convention is transparent to NCO[4]. There are no known pitfalls associated with using

[4] The exception is appending/altering the attributes `x_op`, `y_op`, `z_op`, and `t_op` for variables which have been averaged across space and time dimensions. This feature is scheduled for future inclusion in NCO.

any NCO operator on files adhering to this convention[5]. However, to facilitate maximum user friendliness, NCO does treat certain variables in some CSM files specially. The special functions are not required by the CSM netCDF convention, but experience has shown they do make life easier.

Currently, NCO determines whether a datafile is a CSM output datafile simply by checking whether value of the global attribute `convention` (if it exists) equals '`NCAR-CSM`'. Should `convention` equal '`NCAR-CSM`' in the (first) *input-file*, NCO will attempt to treat certain variables specially, because of their meaning in CSM files. NCO will not average the following variables often found in CSM files: `ntrm`, `ntrn`, `ntrk`, `ndbase`, `nsbase`, `nbdate`, `nbsec`, `mdt`, `mhisf`. These variables contain scalar metadata such as the resolution of the host CSM model and it makes no sense to change their values. Furthermore, the `ncdiff` operator will not attempt to difference the following variables: `gw`, `ORO`, `date`, `datesec`, `hyam`, `hybm`, `hyai`, `hybi`. These variables represent the Gaussian weights, the orography field, time fields, and hybrid pressure coefficients. These are fields which you want to remain unaltered in the differenced file 99% of the time. If you decide you would like any of the above CSM fields processed, you must use `ncrename` to rename them first.

3.15 ARM Conventions

Availability: `ncrcat`
Key switches: None

`ncrcat` has been programmed to recognize ARM (Atmospheric Radiation Measurement Program) data files. If you do not work with ARM data then you may skip this section. ARM data files store time information in two variables, a scalar, `base_time`, and a record variable, `time_offset`. Subtle but serious problems can arise when these type of files are just blindly concatenated. Therefore `ncrcat` has been specially programmed to be able to chain together consecutive ARM *input-files* and produce and an *output-file* which contains the correct time information. Currently, `ncrcat` determines

[5] The CSM convention recommends `time` be stored in the format *time* since *base_time*, e.g., the `units` attribute of `time` might be '`days since 1992-10-8 15:15:42.5 -6:00`'. A problem with this format occurs when using `ncrcat` to concatenate multiple files together, each with a different *base_time*. That is, any `time` values from files following the first file to be concatenated should be corrected to the *base_time* offset specified in the `units` attribute of `time` from the first file. The analogous problem has been fixed in ARM files (see Section 3.15 [ARM Conventions], page 29) and could be fixed for CSM files if there is sufficient lobbying, and if Unidata fixes the UDUNITS package to build out of the box on Linux.

whether a datafile is an ARM datafile simply by testing for the existence of the variables `base_time`, `time_offset`, and the dimension `time`. If these are found in the *input-file* then `ncrcat` will automatically perform two non-standard, but hopefully useful, procedures. First, `ncrcat` will ensure that values of `time_offset` appearing in the *output-file* are relative to the `base_time` appearing in the first *input-file* (and presumably, though not necessarily, also appearing in the *output-file*). Second, if a coordinate variable named `time` is not found in the *input-files*, then `ncrcat` automatically creates the `time` coordinate in the *output-file*. The values of `time` are defined by the ARM convention $time = base\_time + time\_offset$. Thus, if *output-file* contains the `time_offset` variable, it will also contain the `time` coordinate. A short message is added to the `history` global attribute whenever these ARM-specific procedures are executed.

4 Reference manual for all operators

This chapter presents reference pages for each of the operators individually. The operators are presented in alphabetical order. All valid command line switches are included in the syntax statement. Recall that descriptions of many of these command line switches are provided only in Chapter 3 [Common features], page 17, to avoid redundancy. Only options specific to, or most useful with, a particular operator are described in any detail in the sections below.

4.1 `ncatted` netCDF Attribute Editor

SYNTAX

`ncatted [-a` *att_dsc*`] [-a ...] [-D] [-h]`
`[-l path] [-O] [-p path] [-R] [-r]`
input-file `[`*output-file*`]`

DESCRIPTION

`ncatted` edits attributes in a netCDF file. If you are editing attributes then you are spending too much time in the world of metadata, and `ncatted` was written to get you back out as quickly and painlessly as possible. `ncatted` can *append*, *create*, *delete*, *modify*, and *overwrite* attributes (all explained below). Furthermore, `ncatted` allows each editing operation to be applied to every variable in a file, thus saving you time when you want to change attribute conventions throughout a file. `ncatted` interprets character attributes as strings.

Because repeated use of `ncatted` can considerably increase the size of the `history` global attribute (see Section 3.13 [History attribute], page 28), the '`-h`' switch is provided to override automatically appending the command to the `history` global attribute in the *output-file*.

When `ncatted` is used to change the `missing_value` attribute, it changes the associated missing data self-consistently. If the internal floating point representation of a missing value, e.g., 1.0e36, differs between two machines then netCDF files produced on those machines will have incompatible missing values. This allows `ncatted` to change the missing values in files from different machines to a single value so that the files may then be concatenated together, e.g., by `ncrcat`, without losing any information. See Section 3.10 [Missing values], page 26, for more information.

The key to mastering `ncatted` is understanding the meaning of the structure describing the attribute modification, *att_dsc*. Each *att_dsc* contains five elements, which makes using `ncatted` somewhat complicated, but powerful. The *att_dsc* argument structure contains five arguments in the following order:

att_dsc = *att_nm*, *var_nm*, *mode*, *att_type*, *att_val*

att_nm Attribute name. Example: `units`

var_nm Variable name. Example: `pressure`

mode Edit mode abbreviation. Example: `a`. See below for complete listing of valid values of *mode*.

att_type Attribute type abbreviation. Example: `c`. See below for complete listing of valid values of *att_type*.

att_val Attribute value. Example: `pascal`.

There should be no empty space between these five consecutive arguments. The description of these arguments follows in their order of appearance.

The value of *att_nm* is the name of the attribute you want to edit. This meaning of this should be clear to all users of the `ncatted` operator.

The value of *var_nm* is the name of the variable containing the attribute (named *att_nm*) that you want to edit. There are two very important and useful exceptions to this rule. The value of *var_nm* can also be used to direct `ncatted` to edit global attributes, or to repeat the editing operation for every variable in a file. A value of *var_nm* of "global" indicates that *att_nm* refers to a global attribute, rather than a particular variable's attribute. This is the method `ncatted` supports for editing global attributes. If *var_nm* is left blank, on the other hand, then `ncatted` attempts to perform the editing operation on every variable in the file. This option may be convenient to use if you decide to change the conventions you use for describing the data.

The value of *mode* is a single character abbreviation (`a`, `c`, `d`, `m`, or `o`) standing for one of five editing modes:

`a` *Append*. Append value *att_val* to current *var_nm* attribute *att_nm* value *att_val*, if any. If *var_nm* does not have an attribute *att_nm*, there is no effect.

`c` *Create*. Create variable *var_nm* attribute *att_nm* with *att_val* if *att_nm* does not yet exist. If *var_nm* already has an attribute *att_nm*, there is no effect.

`d` *Delete*. Delete current *var_nm* attribute *att_nm*. If *var_nm* does not have an attribute *att_nm*, there is no effect. When *Delete* mode is selected, the *att_type* and *att_val* arguments are superfluous and may be left blank.

`m` *Modify*. Change value of current *var_nm* attribute *att_nm* to value *att_val*. If *var_nm* does not have an attribute *att_nm*, there is no effect.

`o` *Overwrite*. Write attribute *att_nm* with value *att_val* to variable *var_nm*, overwriting existing attribute *att_nm*, if any. This is the default mode.

The value of *att_type* is a single character abbreviation (`f`, `d`, `l`, `s`, `c`, or `b`) standing for one of the six primitive netCDF data types:

`f` *Float*. Value(s) specified in *att_val* will be stored as netCDF intrinsic type NC_FLOAT.

`d` *Double*. Value(s) specified in *att_val* will be stored as netCDF intrinsic type NC_DOUBLE.

`l` *Long*. Value(s) specified in *att_val* will be stored as netCDF intrinsic type NC_LONG.

`s` *Short*. Value(s) specified in *att_val* will be stored as netCDF intrinsic type NC_SHORT.

`c` *Char*. Value(s) specified in *att_val* will be stored as netCDF intrinsic type NC_CHAR.

`b` *Byte*. Value(s) specified in *att_val* will be stored as netCDF intrinsic type NC_BYTE.

The specification of *att_type* is optional in *Delete* mode.

The value of *att_val* is what you want to change attribute *att_nm* to contain. The specification of *att_val* is optional in *Delete* mode. Attribute values for all types besides NC_CHAR must have an attribute length of at least one. Thus *att_val* may be a single value or one-dimensional array of elements of type `att_type`. If the *att_val* is not set or is set to empty space, and the *att_type* is NC_CHAR, e.g., `-a units,T,o,c,""` or `-a units,T,o,c,,` then the corresponding attribute is set to have zero length. When specifying an array of values, it is safest to enclose *att_val* in double or single quotes, e.g., `-a levels,T,o,s,"1,2,3,4"` or `-a levels,T,o,s,'1,2,3,4'`. The quotes are strictly unnecessary around *att_val* except when *att_val* contains characters which would confuse the calling shell, such as spaces, commas, and wildcard characters.

NCO processing of NC_CHAR attributes is a bit like Perl in that it attempts to do you want by default (but this sometimes causes unexpected results if you want unusual data storage). If the *att_type* is NC_CHAR then the argument is interpreted as a string and it may contain C-language escape sequences, e.g., `\n`, which NCO will interpret before writing anything to disk. NCO translates valid escape sequences and stores the appropriate ASCII code instead. Since two byte escape sequences, e.g., `\n`, represent one byte ASCII codes, e.g., ASCII 10 (decimal), the stored string attribute is one byte shorter than the input string length for each embedded escape sequence. The most frequently used C-language escape sequences are `\n` (for linefeed) and `\t` (for horizontal tab). These sequences in particular allow convenient editing of formatted text attributes. The other valid ASCII codes are `\a`, `\b`, `\f`, `\r`, `\v`, and `\\`. See Section 4.6 [ncks netCDF Kitchen Sink], page 45, for more examples of string formatting (with the `ncks` '`-s`' option) with special characters.

Analogous to `printf`, other special characters are also allowed by `ncatted` if they are "protected" by a backslash. The characters ", ', ?, and \ may be input to the shell as \", \', \?, and \\. NCO simply strips away the leading backslash from these characters before editing the attribute. No other characters require protection by a backslash. Backslashes which precede any other character (e.g., `3`, `m`, `$`, `|`, `&`, `@`, `%`, `{`, and `}`) will not be filtered and will be included in the attribute.

Note that the NUL character `\0` which terminates C language strings is assumed and need not be explicitly specified. If `\0` is input, it will not be translated (because it would terminate the string in an additional location). Because of these context-sensitive rules, if wish to use an attribute of type NC_CHAR to store data, rather than text strings, you should use `ncatted` with care.

EXAMPLES

Append the string "Data version 2.0.\n" to the global attribute `history`:

```
ncatted -O -a history,global,a,c,"Data version 2.0\n" in.nc
```

Note the use of embedded C language `printf()`-style escape sequences.

Change the value of the `long_name` attribute for variable `T` from whatever it currently is to "temperature":

```
ncatted -O -a long_name,T,o,c,temperature in.nc
```

Delete all existing `units` attributes:

```
ncatted -O -a units,,d,, in.nc
```

The value of *var_nm* was left blank in order to select all variables in the file. The values of *att_type* and *att_val* were left blank because they are superfluous in *Delete* mode.

Modify all existing `units` attributes to "meter second-1"

```
ncatted -O -a units,,m,c,"meter second-1" in.nc
```

Overwrite the `quanta` attribute of variable `energy` to an array of four integers.

```
ncatted -O -a quanta,energy,o,s,"010,101,111,121" in.nc
```

Demonstrate input of C-language escape sequences (e.g., `\n`) and other special characters (e.g., `\"`)

```
ncatted -h -a special,global,o,c,
'\nDouble quote: \"\nTwo consecutive double quotes: \"\"\n
Single quote: Beyond my shell abilities!\nBackslash: \\\n
Two consecutive backslashes: \\\\\nQuestion mark: \?\n' in.nc
```

Note that the entire attribute is protected from the shell by single quotes. These outer single quotes are necessary for interactive use, but may be omitted in batch scripts.

4.2 `ncdiff` netCDF Differencer

SYNTAX

```
ncdiff [-A] [-C] [-c] [-D dbg]
[-d dim,[min][,[max]]] [-F] [-h] [-l path]
[-O] [-p path] [-R] [-r] [-v var[,...]]
[-x] file_1 file_2 file_3
```

DESCRIPTION

`ncdiff` subtracts variables in *file_2* from the corresponding variables (those with the same name) in *file_1* and stores the results in *file_3*. Variables in *file_2* are *broadcast* to conform to the corresponding variable in *file_1* if necessary. Broadcasting a variable means creating data in non-existing dimensions from the data in existing dimensions. For example, a two dimensional variable in *file_2* can be subtracted from a four, three, or two (but not one or zero) dimensional variable (of the same name) in `file_1`. This functionality allows the user to compute anomalies from the mean. Note that variables in *file_1* are *not* broadcast to conform to the dimensions in *file_2*. Thus, `ncdiff`, the number of dimensions, or *rank*, of any processed variable in *file_1* must be greater than or equal to the rank of the same variable in *file_2*. Furthermore, the size of all dimensions common to both *file_1* and *file_2* must be equal.

When computing anomalies from the mean it is often the case that *file_2* was created by applying an averaging operator to a file with the same dimensions as *file_1*, if not *file_1* itself. In these cases, creating *file_2* with `ncra` rather than `ncwa` will cause the `ncdiff` operation to fail. For concreteness say the record dimension in `file_1` is `time`. If *file_2* were created by averaging *file_1* over the `time` dimension with the `ncra` operator rather than with the `ncwa` operator, then *file_2* will have a `time` dimension of size 1 rather than having no `time` dimension at all [1]. In this case the input files to `ncdiff`, *file_1* and *file_2*, will have unequally sized `time` dimensions which causes `ncdiff` to fail. To prevent this from occuring, use `ncwa` to remove the `time` dimension from *file_2*. An example is given below.

`ncdiff` will never difference coordinate variables or variables of type `NC_CHAR` or `NC_BYTE`. This ensures that coordinates like (e.g., latitude and longitude) are physically meaningful in the output file, *file_3*. This behavior is hardcoded. `ncdiff` applies special rules to some NCAR CSM fields (e.g.,

---

[1] This is because `ncra` collapses the record dimension to a size of 1 (making it a *degenerate* dimension), but does not remove it, while `ncwa` removes all dimensions it averages over. In other words, `ncra` changes the size but not the rank of variables, while `ncwa` changes both the size and the rank of variables.

ORO). See Section 3.14 [NCAR CSM Conventions], page 28 for a complete description.

EXAMPLES

Say files '85_0112.nc' and '86_0112.nc' each contain 12 months of data. Compute the change in the monthly averages from 1985 to 1986:

```
ncdiff 86_0112.nc 85_0112.nc 86m85_0112.nc
```

The following examples demonstrate the broadcasting feature of ncdiff. Say we wish to compute the monthly anomalies of T from the yearly average of T for the year 1985. First we create the 1985 average from the monthly data, which is stored with the record dimension time.

```
ncra 85_0112.nc 85.nc
ncwa -O -a time 85.nc 85.nc
```

The second command, ncwa, gets rid of the time dimension of size 1 that ncra left in '85.nc'. Now none of the variables in '85.nc' has a time dimension. A quicker way to accomplish this is to use ncwa from the beginning:

```
ncwa -a time 85_0112.nc 85.nc
```

We are now ready to use ncdiff to compute the anomalies for 1985:

```
ncdiff -v T 85_0112.nc 85.nc t_anm_85_0112.nc
```

Each of the 12 records in 't_anm_85_0112.nc' now contains the monthly deviation of T from the annual mean of T for each gridpoint.

Say we wish to compute the monthly gridpoint anomalies from the zonal annual mean. A *zonal mean* is a quantity that has been averaged over the longitudinal (or x) direction. First we use ncwa to average over longitudinal direction lon, creating 'xavg_85.nc', the zonal mean of '85.nc'. Then we use ncdiff to subtract the zonal annual means from the monthly gridpoint data:

```
ncwa -a lon 85.nc xavg_85.nc
ncdiff 85_0112.nc xavg_85.nc tx_anm_85_0112.nc
```

Assuming '85_0112.nc' has dimensions time and lon, this example only works if 'xavg_85.nc' has no time or lon dimension.

As a final example, say we have five years of monthly data (i.e., 60 months) stored in '8501_8912.nc' and we wish to create a file which contains the twelve month seasonal cycle of the average monthly anomaly from the five-year mean of this data. The following method is just one permutation of many which will accomplish the same result. First use ncwa to create the file containing the five-year mean:

```
ncwa -a time 8501_8912.nc 8589.nc
```

Next use `ncdiff` to create a file containing the difference of each month's data from the five-year mean:

```
ncdiff 8501_8912.nc 8589.nc t_anm_8501_8912.nc
```

Now use `ncks` to group the five January anomalies together in one file, and use `ncra` to create the average anomaly for all five Januarys. These commands are embedded in a shell loop so they are repeated for all twelve months:

```
foreach idx (01 02 03 04 05 06 07 08 09 10 11 12)
ncks -F -d time,$idx,,12 t_anm_8501_8912.nc foo.$idx
ncra foo.$idx t_anm_8589_$idx.nc
end
```

Note that `ncra` understands the `stride` argument so the two commands inside the loop may be combined into the single command

```
ncra -F -d time,$idx,,12 t_anm_8501_8912.nc foo.$idx
```

Finally, use `ncrcat` to concatenate the 12 average monthly anomaly files into one twelve-record file which contains the entire seasonal cycle of the monthly anomalies:

```
ncrcat t_anm_8589_??.nc t_anm_8589_0112.nc
```

4.3 `ncea` netCDF Ensemble Averager

SYNTAX

```
ncea [-A] [-C] [-c] [-D dbg]
[-d dim,[min][,[max]]] [-F] [-h] [-l path]
[-n loop] [-O] [-p path] [-R] [-r] [-v var[,...]]
[-x] input-files output-file
```

DESCRIPTION

`ncea` performs gridpoint averages of variables across an arbitrary number (an *ensemble*) of input files, with each file receiving an equal weight in the average. Each variable in the *output-file* will be the same size as the same variable in any one of the in the *input-files*, and all *input-files* must be the same size. Whereas `ncra` only performs averages over the record dimension (e.g., time), and weights each record in the record dimension evenly, `ncea` averages entire files, and weights each file evenly. All dimensions, including the record dimension, are treated identically and preserved in the *output-file*. See Section 2.5 [Averaging vs. Concatenating], page 11, for a description of the distinctions between the various averagers and concatenators.

The file is the logical unit of organization for the results of many scientific studies. Often one wishes to generate a file which is the gridpoint average of many separate files. This may be to reduce statistical noise by combining the results of a large number of experiments, or it may simply be a step in a procedure whose goal is to compute anomalies from a mean state. In any case, when one desires to generate a file whose properties are the mean of all the input files, then `ncea` is the operator to use. `ncea` assumes coordinate variable are properties common to all of the experiments and so does not average them across files. Instead, `ncea` copies the values of the coordinate variables from the first input file to the output file.

EXAMPLES

Consider a model experiment which generated five realizations of one year of data, say 1985. You can imagine that the experimenter slightly perturbs the initial conditions of the problem before generating each new solution. Assume each file contains all twelve months (a seasonal cycle) of data and we want to produce a single file containing the ensemble average (mean) seasonal cycle. Here the numeric filename suffix denotes the experiment number (*not* the month):

```
ncea 85_01.nc 85_02.nc 85_03.nc 85_04.nc 85_05.nc 85.nc
ncea 85_0[1-5].nc 85.nc
ncea -n 5,2,1 85_01.nc 85.nc
```

These three commands produce identical answers. See Section 3.1 [Specifying input files], page 17, for an explanation of the distinctions between these methods. The output file, '`85.nc`', is the same size as the inputs files. It contains 12 months of data (which might or might not be stored in the record dimension, depending on the input files), but each value in the output file is the average of the five values in the input files.

In the previous example, the user could have obtained the ensemble average values in a particular spatio-temporal region by adding a hyperslab argument to the command, e.g.,

```
ncea -d time,0,2 -d lat,-23.5,23.5 85_??.nc 85.nc
```

In this case the output file would contain only three slices of data in the *time* dimension. These three slices are the average of the first three slices from the input files. Additionally, only data inside the tropics is included.

4.4 ncecat netCDF Ensemble Concatenator

SYNTAX

```
ncecat [-A] [-C] [-c] [-D dbg]
[-d dim,[min][,[max]]] [-F] [-h] [-l path]
[-n loop] [-O] [-p path] [-R] [-r] [-v var[,...]]
[-x] input-files output-file
```

DESCRIPTION

`ncecat` concatenates an arbitrary number of input files into a single output file. Input files are glued together by creating a record dimension in the output file. Input files must be the same size. Each input file is stored consecutively as a single record in the output file. Thus, the size of the output file is the sum of the sizes of the input files. See Section 2.5 [Averaging vs. Concatenating], page 11, for a description of the distinctions between the various averagers and concatenators.

Consider five realizations, '`85a.nc`', '`85b.nc`', ... '`85e.nc`' of 1985 predictions from the same climate model. Then `ncecat 85?.nc 85_ens.nc` glues the individual realizations together into the single file, '`85_ens.nc`'. If an input variable was dimensioned [`lat`,`lon`], it will have dimensions [`record`,`lat`,`lon`] in the output file. A restriction of `ncecat` is that the hyperslabs of the processed variables must be the same from file to file. Normally this means all the input files are the same size, and contain data on different realizations of the same variables.

EXAMPLES

Consider a model experiment which generated five realizations of one year of data, say 1985. You can imagine that the experimenter slightly perturbs the initial conditions of the problem before generating each new solution. Assume each file contains all twelve months (a seasonal cycle) of data and we want to produce a single file containing all the seasonal cycles. Here the numeric filename suffix denotes the experiment number (*not* the month):

```
ncecat 85_01.nc 85_02.nc 85_03.nc 85_04.nc 85_05.nc 85.nc
ncecat 85_0[1-5].nc 85.nc
ncecat -n 5,2,1 85_01.nc 85.nc
```

These three commands produce identical answers. See Section 3.1 [Specifying input files], page 17, for an explanation of the distinctions between these methods. The output file, '`85.nc`', is five times the size as a single *input-file*. It contains 60 months of data (which might or might not be stored in the record dimension, depending on the input files).

4.5 `ncflint` netCDF File Interpolator

SYNTAX

```
ncflint [-A] [-C] [-c] [-D dbg]
[-d dim,[min][,[max]]] [-F] [-h]
[-i var,val3]
[-l path] [-O] [-p path] [-R] [-r] [-v var[,...]]
[-w wgt1[,wgt2]] [-x] file_1 file_2 file_3
```

DESCRIPTION

`ncflint` creates an output file that is a linear combination of the input files. This linear combination can be a weighted average, a normalized weighted average average, or an interpolation of the input files. Coordinate variables are not acted upon in any case, they are simply copied from *file_1*.

There are two conceptually distinct methods of using `ncflint`. The first method is to specify the weight each input file is to have in the output file. In this method, the value *val3* of a variable in the output file *file_3* is determined from its values *val1* and *val2* in the two input files according to $val3 = wgt1 \times val1 + wgt2 \times val2$. Here at least *wgt1*, and, optionally, *wgt2*, are specified on the command line with the '`-w`' switch. If only *wgt1* is specified then *wgt2* is automatically computed as $wgt2 = 1 - wgt1$. Note that weights larger than 1 are allowed. Thus it is possible to specify $wgt1 = 2$ and $wgt2 = -3$.

The second method of using `ncflint` is specifying the interpolation option with '`-i`'. This is really the inverse of the first method in the following sense. When the user specifies the weights directly, `ncflint` has no work to do besides multiplying the input values by their respective weights and adding the results together to produce the output values. This assumes it is the weights that are known a priori. In another class of cases it is the *arrival value* (i.e., *val3*) of a particular variable *var* that is known a priori. In this case, the implied weights can always be inferred by examining the values of *var* in the input files. This results in one equation in two unknowns, *wgt1* and *wgt2*: $val3 = wgt1 \times val1 + wgt2 \times val2$. Unique determination of the weights requires imposing the additional constraint of normalization on the weights: $wgt1 + wgt2 = 1$. Thus, to use the interpolation option, the user specifies *var* and *val3* with the '`-i`' option. `ncflint` will compute *wgt1* and *wgt2*, and use these weights on all variables to generate the output file. Although *var* may have any number of dimensions in the input files, it must represent a single, scalar value. Thus any dimensions associated with *var* must be *degenerate*, i.e., of size one.

If neither '`-i`' nor '`-w`' is specified on the command line, `ncflint` defaults to weighting each input file equally in the output file. This is equivalent to specifying '-w .5' or '-w .5,.5'. Note that attempting to specify both methods with '-i' and '-w' will result in an error.

ncflint is programmed not to interpolate variables of type NC_CHAR and NC_BYTE. This behavior is hardcoded.

EXAMPLES

Although it has other uses, the interpolation feature was designed to interpolate *file_3* to a time between existing files. Consider input files '85.nc' and '87.nc' containing variables describing the state of a physical system at times time = 85 and time = 87. Assume each file contains its timestamp in the scalar variable time. Then, to linearly interpolate to a file '86.nc' which describes the state of the system at time at time = 86, we would use

```
ncflint -i time,86 85.nc 87.nc 86.nc
```

Say you have observational data covering January and April 1985 in two files named '85_01.nc' and '85_04.nc', respectively. Then you can estimate the values for February and March by interpolating the existing data as follows. Combine '85_01.nc' and '85_04.nc' in a 2:1 ratio to make '85_02.nc':

```
ncflint -w .667 85_01.nc 85_04.nc 85_02.nc
ncflint -w .667,.333 85_01.nc 85_04.nc 85_02.nc
```

Multiply '85.nc' by 3 and by −2 and add them together to make 'tst.nc':

```
ncflint -w 3,-2 85.nc 85.nc tst.nc
```

This is an example of a null operation, so 'tst.nc' should be identical (within machine precision) to '85.nc'.

4.6 `ncks` netCDF Kitchen Sink

SYNTAX

```
ncks [-A] [-a] [-C] [-c] [-D]
[-d dim,[min][,[max]][,[stride]]]
[-F] [-H] [-h] [-l path] [-M] [-m] [-O] [-p path] [-R]
[-r] [-s format] [-u] [-v var[,...]] [-x]
input-file [output-file]
```

DESCRIPTION

`ncks` combines selected features of `ncdump(1)`, `ncextr(1)`, and the nccut and ncpaste specifications into one versatile utility. `ncks` extracts a subset of the data from *input-file* and either prints it as ASCII text to stdout, or writes (or pastes) it to *output-file*, or both.

`ncks` will print netCDF data in ASCII format to `stdout`, like `ncdump(1)`, but with these differences: `ncks` prints data in a tabular format intended to be easy to search for the data you want, one datum per screen line, with all dimension subscripts and coordinate values (if any) preceding the datum. Option '`-s`' allows the user the format the data using C-style format strings.

Options '`-a`', '`-M`', '`-m`', '`-H`', '`-F`', '`-s`', and '`-u`' control the formatted appearance of the data.

`ncks` will extract (and optionally create a new netCDF file comprised of) only selected variable from the input file, like `ncextr(1)` but with these differences: Only variables and coordinates may be specifically included or excluded—all global attributes and any attribute associated with an extracted variable will be copied to the screen and/or output netCDF file. Options '`-c`', '`-C`', '`-v`', and '`-x`' control which variables are extracted.

`ncks` will extract hyperslabs from the specified variables. In fact `ncks` implements the nccut specification exactly. Option '`-d`' controls the hyperslab specification.

Input dimensions that are not associated with any output variable will not appear in the output netCDF. This feature removes superfluous dimensions from a netCDF file.

`ncks` will append variables and attributes from the *input-file* to *output-file* if *output-file* is a pre-existing netCDF file whose relevant dimensions conform to dimension sizes of *input-file*. The append features of `ncks` are intended to provide a rudimentary means of adding data from one netCDF file to another, conforming, netCDF file. When naming conflicts exists between the two files, data in *output-file* is usually overwritten by the corresponding data from *input-file*. Thus it is recommended that the user backup *output-file* in case valuable data is accidentally overwritten.

If *output-file* exists, the user will be queried whether to *overwrite*, *append*, or *exit* the `ncks` call completely. Choosing *overwrite* destroys the existing *output-file* and create an entirely new one from the output of the `ncks` call. Append has differing effects depending on the uniqueness of the variables and attributes output by `ncks`: If a variable or attribute extracted from *input-file* does not have a name conflict with the members of *output-file* then it will be added to *output-file* without overwriting any of the existing contents of *output-file*. In this case the relevant dimensions must agree (conform) between the two files; new dimensions are created in *output-file* as required. When a name conflict occurs, a global attribute from *input-file* will overwrite the corresponding global attribute from *output-file*. If the name conflict occurs for a non-record variable, then the dimensions and type of the variable (and of its coordinate dimensions, if any) must agree (conform) in both files. Then the variable values (and any coordinate dimension values) from *input-file* will overwrite the corresponding variable values (and coordinate dimension values, if any) in *output-file*[1].

Since there can only be one record dimension in a file, the record dimension must have the same name (but not necessarily the same size) in both files if a record dimension variable is to be appended. If the record dimensions are of differing sizes, the record dimension of *output-file* will become the greater of the two record dimension sizes, the record variable from *input-file* will overwrite any counterpart in *output-file* and fill values will be written to any gaps left in the rest of the record variables (I think). In all cases variable attributes in *output-file* are superseded by attributes of the same name from *input-file*, and left alone if there is no name conflict.

Some users may wish to avoid interactive `ncks` queries about whether to overwrite existing data. For example, batch scripts will fail if `ncks` does not receive responses to its queries. Options '`-O`' and '`-A`' are available to force overwriting existing files and variables, respectively.

Options specific to `ncks`:

The following list provides a short summary of the features unique to `ncks`. Features common to many operators are described in Chapter 3 [Common features], page 17.

'`-a`' Do not alphabetize extracted fields. By default, the specified output variables are extracted, printed, and written to disk in

[1] Those familiar with netCDF mechanics might wish to know what is happening here: `ncks` does not attempt to redefine the variable in *output-file* to match its definition in *input-file*, `ncks` merely copies the values of the variable and its coordinate dimensions, if any, from *input-file* to *output-file*.

alphabetical order. This tends to make long output lists easier to search for particular variables. Specifying `-a` results in the variables being extracted, printed, and written to disk in the order in which they were saved in the input file. Thus `-a` retains the original ordering of the variables.

'-d *dim*,[*min*][,[*max*]][,[*stride*]]'
: Add *stride* argument to hyperslabber. For a complete description of the *stride* argument, See Section 3.9 [Stride], page 25.

'-H'
: Print data to screen. Unless otherwise specified (with `-s`), each element of the data hyperslab is printed on a separate line containing the names, indices, and, values, if any, of all of the variables dimensions. The dimension and variable indices refer to the location of the corresponding data element with respect to the variable as stored on disk (i.e., not the hyperslab).

```
% ncks -H -C -v three_dmn_var in.nc
lat[0]=-90 lev[0]=100 lon[0]=0 three_dmn_var[0]=0
lat[0]=-90 lev[0]=100 lon[1]=90 three_dmn_var[1]=1
lat[0]=-90 lev[0]=100 lon[2]=180 three_dmn_var[2]=2
...
lat[1]=90 lev[2]=1000 lon[1]=90 three_dmn_var[21]=21
lat[1]=90 lev[2]=1000 lon[2]=180 three_dmn_var[22]=22
lat[1]=90 lev[2]=1000 lon[3]=270 three_dmn_var[23]=23
```

Printing the same variable with the '-F' option shows the same variable indexed with Fortran conventions

```
% ncks -F -H -C -v three_dmn_var in.nc
lon(1)=0 lev(1)=100 lat(1)=-90 three_dmn_var(1)=0
lon(2)=90 lev(1)=100 lat(1)=-90 three_dmn_var(2)=1
lon(3)=180 lev(1)=100 lat(1)=-90 three_dmn_var(3)=2
...
```

Printing a hyperslab does not affect the variable or dimension indices since these indices are relative to the full variable (as stored in the input file), and the input file has not changed. However, if the hypserslab is saved to an output file and those values are printed, the indices will change:

```
% ncks -H -d lat,90.0 -d lev,1000.0 -v three_dmn_var in.nc out.nc
lat[1]=90 lev[2]=1000 lon[0]=0 three_dmn_var[20]=20
lat[1]=90 lev[2]=1000 lon[1]=90 three_dmn_var[21]=21
lat[1]=90 lev[2]=1000 lon[2]=180 three_dmn_var[22]=22
lat[1]=90 lev[2]=1000 lon[3]=270 three_dmn_var[23]=23
% ncks -H out.nc
lat[0]=90 lev[0]=1000 lon[0]=0 three_dmn_var[0]=20
lat[0]=90 lev[0]=1000 lon[1]=90 three_dmn_var[1]=21
```

```
lat[0]=90 lev[0]=1000 lon[2]=180 three_dmn_var[2]=22
lat[0]=90 lev[0]=1000 lon[3]=270 three_dmn_var[3]=23
```

'-M' Print to screen the global metadata describing the file. This includes file summary information and global attributes.

'-m' Print variable metadata to screen (similar to `ncdump -h`). This displays all the metadata pertaining to each variable, one variable at at time.

'-s *format*'
String format for text output. Accepts C language escape sequences and `printf()` formats.

'-u' Accompany the printing of a variable's values with its units attribute, if it exists.

EXAMPLES

View all data in netCDF '`in.nc`', printed with Fortran indexing conventions:

```
ncks -H -F in.nc
```

Copy the netCDF file '`in.nc`' to file '`out.nc`'.

```
ncks -O in.nc out.nc
```

Now the file '`out.nc`' contains all the data from '`in.nc`'. There are, however, two differences between '`in.nc`' and '`out.nc`'. First, the `history` global attribute (see Section 3.13 [History attribute], page 28) will contain the command used to create '`out.nc`'. Second, the variables in '`out.nc`' will be defined in alphabetical order. Of course the internal storage of variable in a netCDF file should be transparent to the user, but there are cases when alphabetizing a file is useful (see description of `-a` switch).

Print variable `three_dmn_var` from file '`in.nc`' with default notations. Next print `three_dmn_var` as an un-annotated text column. Then print `three_dmn_var` signed with very high precision. Finally, print `three_dmn_var` as a comma-separated list.

```
ncks -H -C -v three_dmn_var in.nc
ncks -s "%f\n" -H -C -v three_dmn_var in.nc
ncks -s "%+16.10f\n" -H -C -v three_dmn_var in.nc
ncks -s "%f, " -H -C -v three_dmn_var in.nc
```

The second and third options are useful when pasting data into text files like reports or papers. See Section 4.1 [ncatted netCDF Attribute Editor], page 32, for more details on string formatting and special characters.

One dimensional arrays of characters stored as netCDF variables are automatically printed as strings, whether or not they are NUL-terminated, e.g.,

```
ncks -v fl_nm in.nc
```

The `%c` formatting code is useful for printing multidimensional arrays of characters representing fixed length strings

```
ncks -H -s "%c" -v fl_nm_arr in.nc
```

Using the `%s` format code on strings which are not NUL-terminated (and thus not technically strings) is likely to result in a core dump.

Create netCDF 'out.nc' containing all variables, and any associated coordinates, except variable `time`, from netCDF 'in.nc':

```
ncks -x -v time in.nc out.nc
```

Extract variables `time` and `pressure` from netCDF 'in.nc'. If 'out.nc' does not exist it will be created. Otherwise the you will be prompted whether to append to or to overwrite 'out.nc':

```
ncks -v time,pressure in.nc out.nc
ncks -C -v time,pressure in.nc out.nc
```

The first version of the command creates an 'out.nc' which contains `time`, `pressure`, and any coordinate variables associated with *pressure*. The 'out.nc' from the second version is guaranteed to contain only two variables `time` and `pressure`.

Create netCDF 'out.nc' containing all variables from file 'in.nc'. Restrict the dimensions of these variables to a hyperslab. Print (with `-H`) the hyperslabs to the screen for good measure. The specified hyperslab is: the fifth value in dimension `time`; the half-open range $lat>0.$ in coordinate `lat`; the half-open range $lon<330.$ in coordinate `lon`; the closed interval $.3<band<.5$ in coordinate `band`; and cross-section closest to 1000. in coordinate `lev`. Note that limits applied to coordinate values are specified with a decimal point, and limits applied to dimension indices do not have a decimal point See Section 3.7 [Hyperslabs], page 22.

```
ncks -H -d time,5 -d lat,,0. -d lon,330., -d band,.3,.5
-d lev,1000. in.nc out.nc
```

Assume the domain of the monotonically increasing longitude coordinate `lon` is $0<lon<360$. Here, `lon` is an example of a wrapped coordinate. `ncks` will extract a hyperslab which crosses the Greenwich meridian simply by specifying the westernmost longitude as *min* and the easternmost longitude as *max*, as follows:

```
ncks -d lon,260.,45. in.nc out.nc
```

For more details See Section 3.8 [Wrapped coordinates], page 24.

4.7 `ncra` netCDF Record Averager

SYNTAX

```
ncra [-A] [-C] [-c] [-D dbg]
[-d dim,[min][,[max]][,[stride]]] [-F] [-h] [-l path]
[-n loop] [-O] [-p path] [-R] [-r] [-v var[,...]]
[-x] input-files output-file
```

DESCRIPTION

`ncra` averages record variables across an arbitrary number of input files. The record dimension is retained as a degenerate (size 1) dimension in the output variables. See Section 2.5 [Averaging vs. Concatenating], page 11, for a description of the distinctions between the various averagers and concatenators.

Input files may vary in size, but each must have a record dimension. The record coordinate, if any, should be monotonic for (or else non-fatal warnings may be generated). Hyperslabs of the record dimension which include more than one file are handled correctly. `ncra` supports the *stride* argument to the '`-d`' hyperslab option for the record dimension only, *stride* is not supported for non-record dimensions.

`ncra` weights each record (e.g., time slice) in the *input-files* equally. `ncra` does not attempt to see if, say, the `time` coordinate is irregularly spaced and thus would require a weighted average in order to be a true time average.

EXAMPLES

Average files '`85.nc`', '`86.nc`', ... '`89.nc`' along the record dimension, and store the results in '`8589.nc`':

```
ncra 85.nc 86.nc 87.nc 88.nc 89.nc 8589.nc
ncra 8[56789].nc 8589.nc
ncra -n 5,2,1 85.nc 8589.nc
```

These three methods produce identical answers. See Section 3.1 [Specifying input files], page 17, for an explanation of the distinctions between these methods.

Assume the files '`85.nc`', '`86.nc`', ... '`89.nc`' each contain a record coordinate *time* of length 12 defined such that the third record in '`86.nc`' contains data from March 1986, etc. NCO knows how to hyperslab the record dimension across files. Thus, to average data from December, 1985 through February, 1986:

```
ncra -d time,11,13 85.nc 86.nc 87.nc 8512_8602.nc
ncra -F -d time,12,14 85.nc 86.nc 87.nc 8512_8602.nc
```

The file '87.nc' is superfluous, but does not cause an error. The '-F' turns on the Fortran (1-based) indexing convention. The following uses the *stride* option to average all the March temperature data from multiple input files into a single output file

```
ncra -F -d time,3,,12 -v temperature 85.nc 86.nc 87.nc 858687_03.nc
```

See Section 3.9 [Stride], page 25, for a description of the *stride* argument.

Assume the *time* coordinate is incrementally numbered such that January, 1985 = 1 and December, 1989 = 60. Assuming '??' only expands to the five desired files, the following averages June, 1985–June, 1989:

```
ncra -d time,6.,54. ??.nc 8506_8906.nc
```

4.8 `ncrcat` netCDF Record Concatenator

SYNTAX

```
ncrcat [-A] [-C] [-c] [-D dbg]
[-d dim,[min][,[max]][,[stride]]] [-F] [-h] [-l path]
[-n loop] [-O] [-p path] [-R] [-r] [-v var[,...]]
[-x] input-files output-file
```

DESCRIPTION

`ncrcat` concatenates record variables across an arbitrary number of input files. The final record dimension is by default the sum of the lengths of the record dimensions in the input files. See Section 2.5 [Averaging vs. Concatenating], page 11, for a description of the distinctions between the various averagers and concatenators.

Input files may vary in size, but each must have a record dimension. The record coordinate, if any, should be monotonic (or else non-fatal warnings may be generated). Hyperslabs of the record dimension which include more than one file are handled correctly. `ncra` supports the *stride* argument to the '`-d`' hyperslab option for the record dimension only, *stride* is not supported for non-record dimensions.

`ncrcat` applies special rules to ARM convention time fields (e.g., `time_offset`). See Section 3.15 [ARM Conventions], page 29 for a complete description.

EXAMPLES

Concatenate files '`85.nc`', '`86.nc`', ... '`89.nc`' along the record dimension, and store the results in '`8589.nc`':

```
ncrcat 85.nc 86.nc 87.nc 88.nc 89.nc 8589.nc
ncrcat 8[56789].nc 8589.nc
ncrcat -n 5,2,1 85.nc 8589.nc
```

These three methods produce identical answers. See Section 3.1 [Specifying input files], page 17, for an explanation of the distinctions between these methods.

Assume the files '`85.nc`', '`86.nc`', ... '`89.nc`' each contain a record coordinate *time* of length 12 defined such that the third record in '`86.nc`' contains data from March 1986, etc. NCO knows how to hyperslab the record dimension across files. Thus, to concatenate data from December, 1985–February, 1986:

```
ncrcat -d time,11,13 85.nc 86.nc 87.nc 8512_8602.nc
ncrcat -F -d time,12,14 85.nc 86.nc 87.nc 8512_8602.nc
```

The file '87.nc' is superfluous, but does not cause an error. The '-F' turns on the Fortran (1-based) indexing convention.

The following uses the *stride* option to concatenate all the March temperature data from multiple input files into a single output file

```
ncrcat -F -d time,3,,12 -v temperature 85.nc 86.nc 87.nc 858687_03.nc
```

See Section 3.9 [Stride], page 25, for a description of the *stride* argument.

Assume the *time* coordinate is incrementally numbered such that January, 1985 = 1 and December, 1989 = 60. Assuming ?? only expands to the five desired files, the following concatenates June, 1985–June, 1989:

```
ncrcat -d time,6.,54. ??.nc 8506_8906.nc
```

4.9 `ncrename` netCDF Renamer

SYNTAX

`ncrename [-a` *old_name,new_name*`] [-a ...] [-D]`
`[-d` *old_name,new_name*`] [-d ...] [-h] [-l path] [-O] [-p path]`
`[-R] [-r] [-v` *old_name,new_name*`] [-v ...]`
input-file `[`*output-file*`]`

DESCRIPTION

`ncrename` renames dimensions, variables, and attributes in a netCDF file. Each object that has a name in the list of old names is renamed using the corresponding name in the list of new names. All the new names must be unique. Every old name must exist in the input file, unless the name is preceded by the character '`.`'. The validity of the old names is not checked prior to the renaming. Thus, if an old name is specified without the the '`.`' prefix and is not present in *input-file*, `ncrename` will abort.

`ncrename` is the exception to the normal rules that the user will be interactively prompted before an existing file is changed, and that a temporary copy of an output file is constructed during the operation. If only *input-file* is specified, then `ncrename` will change the names of the *input-file* in place without prompting and without creating a temporary copy of `input-file`. This is because the renaming operation is considered reversible, i.e., the *new_name* can easily be changed back to *old_name* by using `ncrename` one more time.

Note that renaming a dimension to the name of a dependent variable can be used to invert the relationship between an independent coordinate variable and a dependent variable. In this case, the named dependent variable must be one-dimensional and should have no missing values. Such a variable will become a coordinate variable.

According to the *netCDF User's Guide*, renaming properties in netCDF files does not incur the penalty of recopying the entire file when the *new_name* is shorter than the *old_name*.

OPTIONS

'`-a` *old_name,new_name*'
: Attribute renaming. The old and new names of the attribute are specified by the associated *old_name* and *new_name* values. Global attributes are treated no differently than variable attributes. This option may be specified more than once.

'-d *old_name*,*new_name*'
: Dimension renaming. The old and new names of the dimension are specified by the associated *old_name* and *new_name* values. This option may be specified more than once.

'-v *old_name*,*new_name*'
: Variable renaming. The old and new names of the variable are specified by the associated *old_name* and *new_name* values. This option may be specified more than once.

EXAMPLES

Rename the variable `p` to `pressure` and `t` to `temperature` in netCDF '`in.nc`'. In this case `p` must exist in the input file (or `ncrename` will abort), but the presence of `t` is optional:

```
ncrename -v p,pressure -v .t,temperature in.nc
```

Create netCDF '`out.nc`' identical to '`in.nc`' except the attribute `_FillValue` is changed to `missing_value` (in all variables which possess it) and the global attribute `Zaire` is changed to `Congo`:

```
ncrename -a _FillValue,missing_value -a Zaire,Congo in.nc out.nc
```

4.10 ncwa netCDF Weighted Averager

SYNTAX

```
ncwa [-A] [-a dim[,...]] [-C] [-c] [-D dbg]
[-d dim,[min][,[max]]] [-F] [-h] [-I] [-l path]
[-M val] [-m var] [-N] [-n] [-O] [-o condition]
[-p path] [-R] [-r] [-v var[,...]] [-W] [-w weight]
[-x] input-file output-file
```

DESCRIPTION

`ncwa` averages variables in a single file over arbitrary dimensions, with options to specify weights, masks, and normalization. See Section 2.5 [Averaging vs. Concatenating], page 11, for a description of the distinctions between the various averagers and concatenators. The default behavior of `ncwa` is to arithmetically average every numerical variable over all dimensions and produce a scalar result. To average variables over only a subset of their dimensions, specify these dimensions in a comma-separated list following '`-a`', e.g., '`-a time,lat,lon`'. As with all arithmetic operators, the operation may be restricted to an arbitrary hypserslab by employing the '`-d`' option (see Section 3.7 [Hyperslabs], page 22). `ncwa` also handles values matching the variable's `missing_value` attribute correctly. Moreover, `ncwa` understands how to manipulate user-specified weights, masks, and normalization options. With these options, `ncwa` can compute sophisticated averages (and integrals) from the command line.

mask and *weight*, if specified, are broadcast to conform to the variables being averaged. The rank of variables is reduced by the number of dimensions which they are averaged over. Thus arrays which are one dimensional in the *input-file* and are averaged by `ncwa` appear in the *output-file* as scalars. This allows the user to infer which dimensions may have been averaged. Note that that it is impossible for `ncwa` to make make a *weight* or *mask* of rank W conform to a *var* of rank V if $W > V$. This situation often arises when coordinate variables (which, by definition, are one dimensional) are weighted and averaged. `ncwa` assumes you know this is impossible and so `ncwa` does not attempt to broadcast *weight* or *mask* to conform to *var* in this case, nor does `ncwa` print a warning message telling you this, because it is so common. Specifying *dbg* > *2* does cause `ncwa` to emit warnings in these situations, however.

Non-coordinate variables are always masked and weighted if specified. Coordinate variables, however, may be treated specially. By default, an averaged coordinate variable, e.g., `latitude`, appears in *output-file* averaged the same way as any other variable containing an averaged dimension. In other words, by default `ncwa` weights and masks coordinate variables like all other variables. This design decision was intended to be helpful but for some applications it may be preferable not to weight or mask coordinate variables just like all other variables. Consider the following arguments to `ncwa`: `-a latitude -w lat_wgt -d latitude,0.,90.` where `lat_wgt` is a weight in the `latitude` dimension. Since, by default `ncwa` weights coordinate variables, the value of `latitude` in the *output-file* depends on the weights in *lat_wgt* and is not likely to be 45.—the midpoint latitude of the hyperslab. Option '`-I`' overrides this default behavior and causes `ncwa` not to weight or mask coordinate variables. In the above case, this causes the value of `latitude` in the *output-file* to be 45.—which is a somewhat appealing result. Thus, '`-I`' specifies simple arithmetic averages for the coordinate variables. In the case of latitude, '`-I`' specifies that you prefer to archive the central latitude of the hyperslab over which variables were averaged rather than the area weighted centroid of the hyperslab[1]. Note that the default behavior of ('`-I`') changed on 1998/12/01—before this date the default was not to weight or mask coordinate variables.

`ncwa` defines the generic average of the variable x to be

$$\bar{x}_j = \frac{\sum_{i=1}^{i=N} \mu_i m_i w_i x_i}{\sum_{i=1}^{i=N} \mu_i m_i w_i}$$

where $\bar{x}_j$ is the j'th element of the output hyperslab, x_i is the i'th element of the input hyperslab, μ_i is 1 unless x_i equals the missing value, m_i is 1 unless x_i is masked, and w_i is the weight. This formiddable looking formula represents a simple weighted average whose bells and whistles are all explained below. It is not early to note, however, that when $\mu_i = m_i = w_i = 1$, the generic averaging expression above reduces to a simple arithmetic average. The size J of the output hyperslab for a given variable is the product of all the dimensions of the input variable which are not averaged over. The size N of the input hyperslab contributing to each $\bar{x}_j$ is simply the product of the sizes of all dimensions which are averaged over (i.e., dimensions specified with '`-a`'). Thus N is the number of input elements which *potentially* contribute to each output element. An input element x_i contributes to the output element x_j except in two conditions:

1. x_i equals the *missing value* (see Section 3.10 [Missing values], page 26) for the variable.
2. x_i is located at a point where the masking condition (see [Masking condition], page 59) is false.

Points x_i in either of these two categories do not contribute to x_j, they are ignored. We now define these criteria more rigorously.

[1] If `lat_wgt` contains Gaussian weights then the value of `latitude` in the *output-file* will be the area-weighted centroid of the hyperslab. For the example given, this is about 30 degrees.

Each x_i has an associated Boolean weight μ_i whose value is 0 or 1 (false or true). The value of μ_i is 1 (true) unless x_i equals the *missing value* (see Section 3.10 [Missing values], page 26) for the variable. Thus, for a variable with no `missing_value` attribute, μ_i is always 1. All other NCO arithmetic operators (`ncdiff`, `ncra`, `ncea`, `ncflint`) treat missing values analogously.

Masking condition

Each x_i also has an associated masking weight m_i whose value is 0 or 1 (false or true). The value of m_i is always 1 unless a *mask* is specified (with '`-m`'). As noted above, *mask* is broadcast, if possible, to conform to the variable being averaged. In this case, the value of m_i depends on the *masking condition*. As expected, $m_i = 1$ when the masking condition is *true* and $m_i = 0$ otherwise.

The masking condition has the syntax *mask condition val*. Here *mask* is the name of the masking variable (specified with '`-m`'). The *condition* argument to '`-o`' may be any one of the six arithmetic comparatives: `eq`, `ne`, `gt`, `lt`, `ge`, `le`. These are the Fortran-style character abbreviations for the logical operations $=$, $\neq$, $>$, $<$, $\geq$, $\leq$. The masking condition defaults to `eq` (equality). The *val* argument to '`-M`' is the right hand side of the *masking condition*. Thus for the i'th element of the hyperslab to be averaged, the masking condition is *mask_i condition val*.

Each x_i is also associated with an additional weight w_i whose value may be user-specified. The value of w_i is identically 1 unless the user specifies a *weight* (with '`-w`'). In this case, the value of w_i is determined by the *weight* variable in the *input-files*. As noted above, *weight* is broadcast, if possible, to conform to the variable being averaged.

M is the number of input elements x_i which *actually* contribute to the output element x_j. M is also known as the *tally* and is defined as

$$M = \sum_{i=1}^{i=N} \mu_i m_i$$

M is identical to the denominator of the generic averaging expression except for the omission of the weight w_i. Thus $M = N$ whenever no input points are missing values or are masked. Whether an element contributes to the output, and thus increments M by one, has more to do with the above two criteria (missing value and masking) than with the numeric value of the element per se. For example, $x_i = 0.0$ does contribute to x_j (assuming the `missing_value` attribute is not 0.0 and location i is not masked). The value $x_i = 0.0$ will not change the numerator of the generic averaging expression, but it will change the denominator (unless its weight $w_i = 0.0$ as well).

Normalization

ncwa has one switch which controls the normalization of the averages appearing in the *output-file*. Option '-N' prevents ncwa from dividing the weighted sum of the variable (the numerator in the averaging expression) by the weighted sum of the weights (the denominator in the averaging expression). Thus '-N' tells ncwa to return just the numerator of the above expression.

EXAMPLES

Given file '85_0112.nc':

```
netcdf 85_0112 {
dimensions:
        lat = 64 ;
        lev = 18 ;
        lon = 128 ;
        time = UNLIMITED ; // (12 currently)
variables:
        float lat(lat) ;
        float lev(lev) ;
        float lon(lon) ;
        float time(time) ;
        float scalar_var ;
        float three_dmn_var(lat, lev, lon) ;
        float two_dmn_var(lat, lev) ;
        float mask(lat, lon) ;
        float gw(lat) ;
}
```

Average all variables in 'in.nc' over all dimensions 'out.nc':

```
ncwa in.nc out.nc
```

Every variable in 'in.nc' is reduced to a scalar in 'out.nc' because averaging was performed over all dimensions were averaged over.

Store the zonal (longitudinal) average of 'in.nc' in 'out.nc':

```
ncwa -a lon in.nc out.nc
```

Here the tally is simply the size of lon, or 128.

Compute the meridional (latitudinal) average, with values weighted by the corresponding element of *gw*[2]:

[2] gw stands for *Gaussian weight* in the NCAR climate model.

```
ncwa -w gw -a lat in.nc out.nc
```

Here the tally is simply the size of `lat`, or 64. The sum of the Gaussian weights is 2.0.

Compute the area average over the tropical Pacific:

```
ncwa -w gw -a lat,lon -d lat,-20.,20. -d lon,120.,270.
in.nc out.nc
```

Here the tally is $64 \times 128 = 8192$.

Compute the area average over the globe, but include only points for which $ORO < 0.5$[3]:

```
ncwa -m ORO -M 0.5 -o lt -w gw -a lat,lon in.nc out.nc
```

Assuming 70% of the gridpoints are maritime, then here the tally is $0.70 \times 8192 \approx 5734$.

Compute the global annual average over the maritime tropical Pacific:

```
ncwa -m ORO -M 0.5 -o lt -w gw -a lat,lon,time
-d lat,-20.,20. -d lon,120.,270. in.nc out.nc
```

[3] `ORO` stands for *Orography* in the NCAR climate model. $ORO<0.5$ selects the gridpoints which are covered by ocean.

General Index

'

' (end quote) 34

-

-a 46, 48
-A 27
-c 21
-C 21
-d *dim*,[*min*][,[*max*]] 22, 24, 25, 57
-F 22
-h 28, 32
-H 47
-I 58
-l *output-path* 20
-m 48
-M 48
-n *loop* 13, 17
-O 27
-p *input-path* 17, 20
-r 27
-R 20
-s 48
-u 48
-v *var* 21
-x 21

.

'.rhosts' 19

?

? (question mark) 34

_

_FillValue attribute 56

"

" (double quote) 34

\

\ (backslash) 34
\' (protected end quote) 34
\? (protected question mark) 34
\" (protected double quote) 34
\\ (ASCII \, backslash) 34
\\ (protected backslash) 34
\a (ASCII BEL, bell) 34
\b (ASCII BS, backspace) 34
\f (ASCII FF, formfeed) 34
\n (ASCII LF, linefeed) 34
\n (linefeed) 48
\r (ASCII CR, carriage return) 34
\t (ASCII HT, horizontal tab) 34
\t (horizontal tab) 48
\v (ASCII VT, vertical tab) 34

0

0 (NUL) 35

A

add_offset 15
alphabetize output 46, 48
anomalies 37
ANSI C 6
appending data 45
appending to files 10, 27
appending variables 10
arithmetic operators 26, 57
ARM conventions 29, 53
arrival value 43
ASCII 34
asynchronous file access 19
attribute names 32, 55
attributes 32
attributes, appending 33
attributes, creating 33
attributes, deleting 33
attributes, editing 33
attributes, global 28, 30, 33, 45, 46, 55, 56
attributes, modifying 33
attributes, overwriting 33
averaging data 26, 40, 51, 57

B

base_time 29
broadcasting variables 37, 57
buffering 15

bugs, reporting 8

C

C index convention 22
C_format 15
CCM Processor 17, 51, 53
characters, special 34
climate modeling 9
compatability 5
concatenation 10, 42, 53
coordinate limits 22
coordinate variable 58
Cray 13
CSM conventions 28, 37
`Cygwin` 7

D

data safety 9, 55
data, missing 26, 32
`date` 28
`datesec` 28
debug_level 14
debugging 14
degenerate dimensions 37, 43, 51
differencing data 37
dimension limits 22
dimension names 55
documentation 5
dynamic linking 7

E

editing attributes 32
ensemble 12, 40
ensemble average 40
ensemble concatenation 42
error tolerance 9
execution time 7, 10, 15, 26, 55

F

file deletion 20
file removal 20
file retention 20
files, multiple 17
files, numerous input 13
force append 27
force overwrite 27
foreword 1
fortran 51, 53
Fortran index convention 22
FORTRAN_format 15
`ftp` 7, 19

G

Gaussian weights 60
GCM 9
global attributes 28, 30, 33, 45, 46, 55, 56
globbing 17, 51, 53
'`GNUmakefile`' 7
`gw` 28, 60

H

HDF 8
Hierarchical Data Format 8
`history` attribute 28, 30, 32, 48
HTML 5
hyperslab 22, 51, 53, 57

I

`ilimit` 13
index conventions 22
Info 5
input-path 17, 20
installation 5
interpolation 43
introduction 5

K

kitchen sink 45

L

large files 13
`LD_LIBRARY_PATH` 7
libraries 7
longitude 24

M

masked average 57
masking condition 59
Mass Store System 19
memory requirements 14, 21
merging files 10, 45
missing values 26, 32, 58

missing_value attribute 26, 32, 56
monotonic coordinates 15
msrcp 19
msread 19
MSS 19
multi-file operators 17

N

NC_BYTE 23
NC_CHAR 23
NCAR 9
NCAR CSM conventions 28, 37
NCAR MSS 19
ncatted 26, 28, 32
ncdiff 27, 37
ncea 12, 27, 40
ncecat 11, 42
ncflint 27, 43
ncks 45
NCL 9
NCO availability 5
NCO homepage 5
NCO User's Guide 5
ncra 12, 27, 51
ncrcat 11, 53
ncrename 55
ncwa 12, 27, 57
netCDF 5
netCDF 2.x 8
netCDF 3.x 8
NETCDF2_ONLY 8
NINTAP 17, 51, 53
normalization 60
nrnet 19
NUL 35
NUL-termination 35

O

on-line documentation 5
operator speed 7, 10, 15, 26, 55
operators 3
ORO 28, 61
OS 5
output-path 20
overwriting files 10, 27

P

pasting variables 10
performance 6, 7, 10, 15, 26, 55
Perl 9, 34
philosophy 9
portability 5
preprocessor tokens 6, 7
printf() 34, 48
printing files contents 45
printing variables 45
Processor 51, 53
Processor, CCM 17

R

rank 37, 57
rcp 7, 19
RCS 27
record average 51
record concatenation 53
regular expressions 17
remote files 7, 19
renaming attributes 55
renaming dimensions 55
renaming variables 55
reporting bugs 8
running average 51

S

safeguards 9, 55
scale_format 15
scp 7, 19
server 13
signedness 15
sort alphabetically 46, 48
source code 5
special characters 34
speed 7, 10, 14, 15, 26, 55
static linking 7
stride 25, 47, 51, 52, 53, 54
strings 35
stub 20
subtraction 37
summary 3
swap space 13
synchronous file access 19

T

temporary output files 9, 55
TEXinfo 5
`time` 28, 29
`time_offset` 29
timestamp 28

U

UNICOS 13
UNIX 5
`UNIX` 7
UNIX 17
URL 19
`USE_FORTRAN_ARITHMETIC` 6, 7
User's Guide 5

V

variable names 55
version 27

W

weighted average 57
`WIN32` 7
Windows NT 5, 6
wrapped coordinates 23, 24, 49
wrapped filenames 18
WWW documentation 5

Y

Yorick 9, 15

Table of Contents

Foreword **1**

Summary **3**

1 Introduction **5**

1.1 Availability 5
1.2 Operating systems compatible with NCO 5
1.2.1 Compiling NCO for Windows NT 6
1.3 Libraries 7
1.4 netCDF 2.x vs. 3.x 8
1.5 Reporting bugs 8

2 Operator Strategies **9**

2.1 NCO operator philosophy 9
2.2 Climate model paradigm 9
2.3 Temporary output files 9
2.4 Appending variables to a file 10
2.5 Averagers vs. Concatenators 11
2.5.1 Concatenators `ncrcat` and `ncecat` 11
2.5.2 Averagers `ncea`, `ncra`, and `ncwa` 12
2.6 Working with large numbers of input files 13
2.7 Working with large files 13
2.8 Approximate NCO memory requirements 14
2.9 Performance limitations of the operators 14

3 Features common to most operators **17**

3.1 Specifying input files 17
3.2 Accessing files stored remotely 19
3.3 Retention of remotely retrieved files 20
3.4 Including/Excluding specific variables 21
3.5 Including/Excluding coordinate variables 21
3.6 C & Fortran index conventions 22
3.7 Hyperslabs 22
3.8 Wrapped coordinates 24
3.9 Stride 25
3.10 Missing values 26
3.11 Suppressing interactive prompts 27
3.12 Operator version 27
3.13 History attribute 28
3.14 NCAR CSM Conventions 28

3.15 ARM Conventions 29

4 Reference manual for all operators 31

4.1 ncatted netCDF Attribute Editor 32
4.2 ncdiff netCDF Differencer 37
4.3 ncea netCDF Ensemble Averager 40
4.4 ncecat netCDF Ensemble Concatenator 42
4.5 ncflint netCDF File Interpolator 43
4.6 ncks netCDF Kitchen Sink 45
Options specific to ncks: 46
4.7 ncra netCDF Record Averager 51
4.8 ncrcat netCDF Record Concatenator 53
4.9 ncrename netCDF Renamer 55
4.10 ncwa netCDF Weighted Averager 57
Masking condition 59
Normalization 60

General Index 63