NCO User's Guide

A suite of netCDF operators
Edition 3.9.2, for NCO Version 3.9.2
August 2007

by Charlie Zender
Department of Earth System Science
University of California, 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
3200 Croul Hall
University of California, Irvine
Irvine, CA 92697-3100 USA

The original author of this software, Charlie Zender, wants to improve it with the help of your suggestions, improvements, bug-reports, and patches.
Charlie Zender <surname at uci dot edu> (yes, my surname is zender)
Department of Earth System Science
3200 Croul Hall
University of California, Irvine
Irvine, CA 92697-3100

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

Major feature improvements entitle me to write another Foreword. In the last five years a lot of work has been done refining NCO. NCO is now an open source project and appears to be much healthier for it. The list of illustrious institutions which do not endorse NCO continues to grow, and now includes UCI.

Charlie Zender
October 2000
Irvine, California

The most remarkable advances in NCO capabilities in the last few years are due to contributions from the Open Source community. Especially noteworthy are the contributions of Henry Butowsky and Rorik Peterson.

Charlie Zender
January 2003
Irvine, California

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 executed at the shell-level like, e.g., `ls` or `mkdir`. The operators take netCDF files (including HDF5 files constructed using the netCDF API) 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 stems from their origin, though 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 `http://sf.net/projects/nco` and from `http://dust.ess.uci.edu/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 DVI, Postscript, and PDF 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 `http://nco.sf.net`. The HTML version of the *User's Guide* is also available online through the World Wide Web at URL `http://nco.sf.net/nco.html`. To build and use NCO, you must have netCDF installed. The netCDF homepage is `http://www.unidata.ucar.edu/packages/netcdf`.

New NCO releases are announced on the netCDF list and on the `nco-announce` mailing list `http://lists.sf.net/mailman/listinfo/nco-announce`.

1.2 Operating systems compatible with NCO

NCO has been successfully ported and tested and is known to work on the following 32- and 64-bit platforms: IBM AIX 4.x, 5.x, FreeBSD 4.x, GNU/Linux 2.x, LinuxPPC, LinuxAlpha, LinuxARM, LinuxSparc64, SGI IRIX 5.x and 6.x, MacOS X 10.x, NEC Super-UX 10.x, DEC OSF, Sun SunOS 4.1.x, Solaris 2.x, Cray UNICOS 8.x–10.x, and MS Windows95 and all later versions. 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 library (and, as of 2003, the UDUnits library). Unidata has shown a commitment to maintaining netCDF and UDUnits on all popular UNIX platforms, and is moving towards full support for the Microsoft Windows operating system (OS). Given this, the only difficulty in implementing NCO on a particular platform is standardization of various C and Fortran interface and system calls. NCO code is tested for ANSI compliance by compiling with C compilers including

[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.

those from GNU ('`gcc -std=c99 -pedantic -D_BSD_SOURCE -D_POSIX_SOURCE' -Wall`)[2], Comeau Computing ('`como --c99`'), Cray ('`cc`'), HP/Compaq/DEC ('`cc`'), IBM ('`xlc -c -qlanglvl=extc99`'), Intel ('`icc -std=c99`'), NEC ('`cc`'), PathScale (QLogic) ('`pathcc -std=c99`'), PGI ('`pgcc -c9x`'), SGI ('`cc -c99`'), and Sun ('`cc`'). NCO (all commands and the `libnco` library) and the C++ interface to netCDF (called `libnco_c++`) comply with the ISO C++ standards as implemented by Comeau Computing ('`como`'), Cray ('`CC`'), GNU ('`g++ -Wall`'), HP/Compaq/DEC ('`cxx`'), IBM ('`xlC`'), Intel ('`icc`'), NEC ('`c++`'), PathScale (Qlogic) ('`pathCC`'), PGI ('`pgCC`'), SGI ('`CC -LANG:std`'), and Sun ('`CC -LANG:std`'). See '`nco/bld/Makefile`' and '`nco/src/nco_c++/Makefile.old`' for more details and exact settings.

Until recently (and not even yet), ANSI-compliant has meant compliance with the 1989 ISO C-standard, usually called C89 (with minor revisions made in 1994 and 1995). C89 lacks variable-size arrays, restricted pointers, some useful `printf` formats, and many mathematical special functions. These are valuable features of C99, the 1999 ISO C-standard. NCO is C99-compliant where possible and C89-compliant where necessary. Certain branches in the code are required to satisfy the native SGI and SunOS C compilers, which are strictly ANSI C89 compliant, and cannot benefit from C99 features. However, C99 features are fully supported by modern AIX, GNU, Intel, NEC, Solaris, and UNICOS compilers. NCO requires a C99-compliant compiler as of NCO version 2.9.8, released in August, 2004.

The most time-intensive portion of NCO execution is spent in arithmetic operations, e.g., multiplication, averaging, subtraction. These operations were performed in Fortran by default until August, 1999. This was a design decision based on the relative speed of Fortran-based object code vs. C-based object code in late 1994. C compiler vectorization capabilities have dramatically improved since 1994. We have accordingly replaced all Fortran subroutines with C functions. This greatly simplifies the task of building NCO on nominally unsupported platforms. As of August 1999, NCO built entirely in C by default. This allowed NCO to compile on any machine with an ANSI C compiler. In August 2004, the first C99 feature, the `restrict` type qualifier, entered NCO in version 2.9.8. C compilers can obtain better performance with C99 restricted pointers since they inform the compiler when it may make Fortran-like assumptions regarding pointer contents alteration. Subsequently, NCO requires a C99 compiler to build correctly[3].

In June 2005, NCO version 3.0.1 began to take advantage of C99 mathematical special functions. These include the standarized gamma function (called `tgamma()` for "true gamma"). NCO automagically takes advantage of some GNU Compiler Collection (GCC) extensions to ANSI C.

As of July 2000 and NCO version 1.2, NCO no longer performs arithmetic operations in Fortran. We decided to sacrifice executable speed for code maintainability. Since no objective statistics were ever performed to quantify the difference in speed between the Fortran and C code, the performance penalty incurred by this decision is unknown. Supporting Fortran involves maintaining two sets of routines for every arithmetic operation. The `USE_FORTRAN_ARITHMETIC` flag is still retained in the '`Makefile`'. The file containing the Fortran

[2] The '`_BSD_SOURCE`' token is required on some Linux platforms where `gcc` dislikes the network header files like '`netinet/in.h`').

[3] NCO may still build with an ANSI or ISO C89 or C94/95-compliant compiler if the C pre-processor undefines the `restrict` type qualifier, e.g., by invoking the compiler with '`-Drestrict=''`'.

code, '`nco_fortran.F`', has been deprecated but a volunteer (Dr. Frankenstein?) could resurrect it. If you would like to volunteer to maintain '`nco_fortran.F`' please contact me.

1.2.1 Compiling NCO for Microsoft Windows OS

NCO has been successfully ported and tested on the Microsoft Windows (95/98/NT/2000/XP) operating systems. The switches necessary to accomplish this are included in the standard distribution of NCO. Using the freely available Cygwin (formerly gnu-win32) development environment[4], the compilation process is very similar to installing NCO on a UNIX system. Set the `PVM_ARCH` preprocessor token to `WIN32`. Note that defining `WIN32` has the side effect of disabling Internet features of NCO (see below). NCO should now build like it does on UNIX.

The least portable section of the code is the use of standard UNIX and Internet protocols (e.g., `ftp`, `rcp`, `scp`, `sftp`, `getuid`, `gethostname`, and header files '`<arpa/nameser.h>`' and '`<resolv.h>`'). Fortunately, these UNIX-y calls are only invoked by the single NCO subroutine which is responsible for retrieving files stored on remote systems (see Section 3.7 [Remote storage], page 24). In order to support NCO on the Microsoft Windows platforms, this single feature was disabled (on Windows OS 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 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 NCO operators will fail with a message that they cannot find a dynamically loaded (aka *shared object* or '`.so`') library. This will produce 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[5], or whether you simply need to alter your library search path. As a final remedy, you may re-compile and install NCO with all operators statically linked.

1.4 netCDF2/3/4 and HDF4/5 Support

netCDF version 2 was released in 1993. NCO (specifically `ncks`) began soon after this in 1994. netCDF 3.0 was released in 1996, and we were eager to reap the performance advantages of the newer netCDF implementation. One netCDF3 interface call (`nc_inq_libvers`) was added to NCO in January, 1998, to aid in maintainance and debugging. In

[4] The Cygwin package is available from
`http://sourceware.redhat.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.

[5] 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'`.

March, 2001, the final conversion of NCO to netCDF3 was completed (coincidentally on the same day netCDF 3.5 was released). NCO versions 2.0 and higher are built with the `-DNO_NETCDF_2` flag to ensure no netCDF2 interface calls are used.

However, the ability to compile NCO with only netCDF2 calls is worth maintaining because HDF version 4[6] (available from HDF) supports only the netCDF2 library calls (see http://hdf.ncsa.uiuc.edu/UG41r3_html/SDS_SD.fm12.html#47784). Note that there are multiple versions of HDF. Currently HDF version 4.x supports netCDF2 and thus NCO version 1.2.x. If NCO version 1.2.x (or earlier) is built with only netCDF2 calls then all NCO operators should work with HDF4 files as well as netCDF files[7]. The preprocessor token `NETCDF2_ONLY` exists in NCO version 1.2.x to eliminate all netCDF3 calls. Only versions of NCO numbered 1.2.x and earlier have this capability. The NCO 1.2.x branch will be maintained with bugfixes only (no new features) until HDF begins to fully support the netCDF3 interface (which is employed by NCO 2.x). If, at compilation time, `NETCDF2_ONLY` is defined, then NCO version 1.2.x will not use any netCDF3 calls and, if linked properly, the resulting NCO operators will work with HDF4 files. The '`Makefile`' supplied with NCO 1.2.x is written to simplify building in this HDF capability. When NCO is built with `make HDF4=Y`, the '`Makefile`' sets all required preprocessor flags and library links to build with the HDF4 libraries (which are assumed to reside under `/usr/local/hdf4`, edit the '`Makefile`' to suit your installation).

HDF version 5 became available in 1999, but did not support netCDF (or, for that matter, Fortran) as of December 1999. By early 2001, HDF5 did support Fortran90. However, support for netCDF4 in HDF5 is incomplete. Much of the HDF5-netCDF interface is complete, however, and it may be separately downloaded from the netCDF4 website. We are eager for HDF5 to complete netCDF support. This is scheduled to occur sometime in 2007, with the releases of HDF version 1.8 and netCDF version 4, which are collaborations between Unidata and NCSA. NCO version 3.0.3 added support for reading/writing netCDF4-formatted HDF5 files in October, 2005. See Section 3.9 [Selecting Output File Format], page 28 for more details.

NCO version 3.9.0 added full support for all netCDF4 atomic data types in May, 2007. Support for netCDF4 features will be incremental, i.e., we will add one netCDF4 feature at a time. You must build NCO with netCDF4 to obtain this support.

The main netCDF4 features that NCO currently supports are the new atomic data types and Lempel-Ziv compression. The new atomic data types are `NC_UBYTE`, `NC_SHORT`, `NC_UINT`, `NC_INT64`, and `NC_UINT64`. Eight-byte integer support is especially useful improvement from netCDF3. All NCO operators support these types, e.g., `ncks` copies and prints them, `ncra` averages them, and `ncap2` processes algebraic scripts with them.

Lempel-Ziv deflation is a lossless compression technique. See Section 4.7 [ncks netCDF Kitchen Sink], page 82 for more details.

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

[7] One must link the NCO code to the HDF4 MFHDF library instead of the usual netCDF library. Does '`MF`' stands for Mike Folk? Perhaps. In any case, the MFHDF library only supports netCDF2 calls. Thus I will try to keep this capability in NCO as long as it is not too much trouble.

netCDF4-enabled NCO handles netCDF3 files without change. In addition, it automagically handles netCDF4 (HDF5) files: If you feed NCO netCDF3 files, it produces netCDF3 output. If you feed NCO netCDF4 files, it produces netCDF4 output. Use the handy-dandy '`-4`' switch to request netCDF4 output from netCDF3 input, i.e., to convert netCDF3 to netCDF4. See Section 3.9 [Selecting Output File Format], page 28 for more details.

Use appropriate caution while netCDF4 is beta software. Problems with netCDF4 and HDF libraries are still being fixed. NCO support for netCDF4 atomic types is relatively untested. Binary NCO distributions (RPMs and debs) still use netCDF3.

For now you must build NCO from source to get netCDF4 support. Typically, one specifies the root of the netCDF4-beta installation directory. Do this with the `NETCDF4_ROOT` variable. Then use your preferred NCO build mechanism, e.g.,

```
export NETCDF4_ROOT=/usr/local/netcdf4 # Set netCDF4 location
cd ~/nco;./configure --enable-netcdf4  # Configure mechanism -or-
cd ~/nco/bld;./make NETCDF4=Y allinone # Old Makefile mechanism
```

Our short term goal is to track the netCDF4-beta releases, keep the new netCDF4 atomic type support working, and iron out any problems. Our long term goal is to utilize more of the extensive new netCDF4 feature set. The next major netCDF4 feature we are likely to utilize is parallel I/O. We will enable this in the MPI netCDF operators.

1.5 Help Requests and Bug Reports

We generally receive three categories of mail from users: help requests, bug reports, and feature requests. Notes saying the equivalent of "Hey, NCO continues to work great and it saves me more time everyday than it took to write this note" are a distant fourth.

There is a different protocol for each type of request. The preferred etiquette for all communications is via NCO Project Forums. Do not contact project members via personal e-mail unless your request comes with money or you have damaging information about our personal lives. *Please use the Forums*—they preserve a record of the questions and answers so that others can learn from our exchange. Also, since NCO is government-funded, this record helps us provide program officers with information they need to evaluate our project.

Before posting to the NCO forums described below, you might first register your name and email address with SourceForge.net or else all of your postings will be attributed to "nobody". Once registered you may choose to "monitor" any forum and to receive (or not) email when there are any postings including responses to your questions. We usually reply to the forum message, not to the original poster.

If you want us to include a new feature in NCO, check first to see if that feature is already on the TODO list. If it is, why not implement that feature yourself and send us the patch? If the feature is not yet on the list, then send a note to the NCO Discussion forum.

Read the manual before reporting a bug or posting a help request. Sending questions whose answers are not in the manual is the best way to motivate us to write more documentation. We would also like to accentuate the contrapositive of this statement. If you think you have found a real bug *the most helpful thing you can do is simplify the problem to*

a manageable size and then report it. The first thing to do is to make sure you are running the latest publicly released version of NCO.

Once you have read the manual, if you are still unable to get NCO to perform a documented function, submit a help request. Follow the same procedure as described below for reporting bugs (after all, it might be a bug). That is, describe what you are trying to do, and include the complete commands (run with '`-D 5`'), error messages, and version of NCO (with '`-r`'). Post your help request to the NCO Help forum.

If you think you used the right command when NCO misbehaves, then you might have found a bug. Incorrect numerical answers are the highest priority. We usually fix those within one or two days. Core dumps and sementation violations receive lower priority. They are always fixed, eventually.

How do you simplify a problem that reveal a bug? Cut out extraneous variables, dimensions, and metadata from the offending files and re-run the command until it no longer breaks. Then back up one step and report the problem. Usually the file(s) will be very small, i.e., one variable with one or two small dimensions ought to suffice. Run the operator with '`-r`' and then run the command with '`-D 5`' to increase the verbosity of the debugging output. It is very important that your report contain the exact error messages and compile-time environment. Include a copy of your sample input file, or place one on a publically accessible location, of the file(s). Post the full bug report to the NCO Project buglist.

Build failures count as bugs. Our limited machine access means we cannot fix all build failures. The information we need to diagnose, and often fix, build failures are the three files output by GNU build tools, '`nco.config.log.${GNU_TRP}.foo`', '`nco.configure.${GNU_TRP}.foo`', and '`nco.make.${GNU_TRP}.foo`'. The file '`configure.eg`' shows how to produce these files. Here `${GNU_TRP}` is the "GNU architecture triplet", the *chip-vendor-OS* string returned by '`config.guess`'. Please send us your improvements to the examples supplied in '`configure.eg`'. The regressions archive at `http://dust.ess.uci.edu/nco/rgr` contains the build output from our standard test systems. You may find you can solve the build problem yourself by examining the differences between these files and your own.

2 Operator Strategies

2.1 Philosophy

The main design goal is command line operators which perform useful, scriptable 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 2.9 [Memory Requirements], page 17.

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 are 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".

[1] The `ncrename` operator is an exception to this rule. See Section 4.11 [ncrename netCDF Renamer], page 99.

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 slows 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

Adding variables from one file to another is often desirable. 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 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 4.7 [ncks netCDF Kitchen Sink], page 82.

The simplest way to create the union of two files is

```
ncks -A fl_1.nc fl_2.nc
```

This puts the contents of '`fl_1.nc`' into '`fl_2.nc`'. The '`-A`' is optional. On output, '`fl_2.nc`' is the union of the input files, regardless of whether they share dimensions and variables, or are completely disjoint. The append fails if the input files have differently named record dimensions (since netCDF supports only one), or have dimensions of the same name but different sizes.

2.5 Simple Arithmetic and Interpolation

Users comfortable with NCO semantics may find it easier to perform some simple mathematical operations in NCO rather than higher level languages. `ncbo` (see Section 4.3 [ncbo netCDF Binary Operator], page 70) does file addition, subtraction, multiplication, division, and broadcasting. `ncflint` (see Section 4.6 [ncflint netCDF File Interpolator], page 79) does file addition, subtraction, multiplication and interpolation. Sequences of these commands can accomplish simple but powerful operations from the command line.

---

[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*?

2.6 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.

2.6.1 Concatenators `ncrcat` and `ncecat`

Joining independent files together along a record dimension is called *concatenation*. `ncrcat` is designed for concatenating record variables, while `ncecat` is designed for concatenating fixed length variables. Consider five 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 five 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 4.10 [ncrcat netCDF Record Concatenator], page 97, for a complete description of `ncrcat`.

However, suppose the annual files have no record variable, and thus their data are 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 is converted to a fixed-length dimension of the same name and a new record dimension (named `record`) is 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 4.5 [ncecat netCDF Ensemble Concatenator], page 77, for a complete description of `ncecat`.

`ncpdq` makes it possible to concatenate files along any dimension, not just the record dimension. First, use `ncpdq` to convert the dimension to be concatenated (i.e., extended with data from other files) into the record dimension. Second, use `ncrcat` to concatenate these files. Finally, if desirable, use `ncpdq` to revert to the original dimensionality. As a

[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.

concrete example, say that files 'x_01.nc', 'x_02.nc', ... 'x_10.nc' contain time-evolving datasets from spatially adjacent regions. The time and spatial coordinates are time and x, respectively. Initially the record dimension is time. Our goal is to create a single file that contains joins all the spatially adjacent regions into one single time-evolving dataset.

```
for idx in 01 02 03 04 05 06 07 08 09 10; do # Bourne Shell
  ncpdq -a x,time x_${idx}.nc foo_${idx}.nc # Make x record dimension
done
ncrcat foo_??.nc out.nc       # Concatenate along x
ncpdq -a time,x out.nc out.nc # Revert to time as record dimension
```

Note that ncrcat will not concatenate fixed-length variables, whereas ncecat concatenates both fixed-length and record variables along a new record variable. To conserve system memory, use ncrcat where possible.

2.6.2 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 4.4 [ncea netCDF Ensemble Averager], page 75, 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 4.9 [ncra netCDF Record Averager], page 95, for a complete description of ncra).

2.6.3 Interpolator ncflint

ncflint can interpolate data between or two files. Since no other operators have this ability, the description of interpolation is given fully on the ncflint reference page (see Section 4.6 [ncflint netCDF File Interpolator], page 79). Note that this capability also allows ncflint to linearly rescale any data in a netCDF file, e.g., to convert between differing units.

2.7 Large Numbers of Files

Occasionally one desires to digest (i.e., concatenate or average) hundreds or thousands of input files. Unfortunately, data archives (e.g., NASA EOSDIS) may not name netCDF files in a format understood by the '-n *loop*' switch (see Section 3.5 [Specifying Input Files], page 22) that automagically generates arbitrary numbers of input filenames. The '-n *loop*' switch has the virtue of being concise, and of minimizing the command line. This helps keeps output file small since the command line is stored as metadata in the history attribute (see Section 3.25 [History Attribute], page 50). However, the '-n *loop*' switch is useless when there is no simple, arithmetic pattern to the input filenames (e.g., 'h00001.nc', 'h00002.nc', ... 'h90210.nc'). Moreover, filename globbing does not work when the input files are too

numerous or their names are too lengthy (when strung together as a single argument) to be passed by the calling shell to the NCO operator[6]. 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. There are (at least) three alternative methods of specifying the input filenames to NCO in environment-limited situations.

The recommended method for sending very large numbers (hundreds or more, typically) of input filenames to the multi-file operators is to pass the filenames with the UNIX *standard input* feature, aka `stdin`:

```
# Pipe large numbers of filenames to stdin
/bin/ls | grep ${CASEID}_'......'.nc | ncecat -o foo.nc
```

This method avoids all constraints on command line size imposed by the operating system. A drawback to this method is that the `history` attribute (see Section 3.25 [History Attribute], page 50) does not record the name of any input files since the names were not passed on the command line. This makes determining the data provenance at a later date difficult. To remedy this situation, multi-file operators store the number of input files in the `nco_input_file_number` global attribute and the input file list itself in the `nco_input_file_list` global attribute (see Section 3.26 [File List Attributes], page 51). Although this does not preserve the exact command used to generate the file, it does retains all the information required to reconstruct the command and determine the data provenance.

A second option is to use the UNIX `xargs` command. This simple example selects as input to `xargs` all the filenames in the current directory that match a given pattern. For illustration, consider a user trying to average millions of files which each have a six character filename. If the shell buffer can not hold the results of the corresponding globbing operator, '`??????.nc`', then the filename globbing technique will fail. Instead we express the filename pattern as an extended regular expression, '`......\.nc`' (see Section 3.11 [Subsetting Variables], page 30). We use `grep` to filter the directory listing for this pattern and to pipe the results to `xargs` which, in turn, passes the matching filenames to an NCO multi-file operator, e.g., `ncecat`.

```
# Use xargs to transfer filenames on the command line
/bin/ls | grep ${CASEID}_'......'.nc | xargs -x ncecat -o foo.nc
```

The single quotes protect the only sensitive parts of the extended regular expression (the `grep` argument), and allow shell interpolation (the `${CASEID}` variable substitution) to proceed unhindered on the rest of the command. `xargs` uses the UNIX pipe feature to append the suitably filtered input file list to the end of the `ncecat` command options. The `-o foo.nc` switch ensures that the input files supplied by `xargs` are not confused with the output file name. `xargs` does, unfortunately, have its own limit (usually about 20,000 characters) on the size of command lines it can pass. Give `xargs` the '`-x`' switch to ensure it dies if it reaches this internal limit. When this occurs, use either the `stdin` method above, or the symbolic link presented next.

[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 (including results of file globbing) exceeding 4096 characters.

Even when its internal limits have not been reached, the `xargs` technique may not be sophisticated enough to handle all situations. A full scripting language like Perl can handle any level of complexity of filtering input filenames, and any number of filenames. The technique of last resort is to write a script that creates symbolic links between the irregular input filenames and a set of regular, arithmetic filenames that the '`-n loop`' switch understands. For example, the following Perl script a monotonically enumerated symbolic link to up to one million '`.nc`' files in a directory. If there are 999,999 netCDF files present, the links are named '`000001.nc`' to '`999999.nc`':

```
# Create enumerated symbolic links
/bin/ls | grep \.nc | perl -e \
'$idx=1;while(<STDIN>){chop;symlink $_,sprintf("%06d.nc",$idx++);}'
ncecat -n 999999,6,1 000001.nc foo.nc
# Remove symbolic links when finished
/bin/rm ??????.nc
```

The '`-n loop`' option tells the NCO operator to automatically generate the filnames of the symbolic links. This circumvents any OS and shell limits on command line size. The symbolic links are easily removed once NCO is finished. One drawback to this method is that the `history` attribute (see Section 3.25 [History Attribute], page 50) retains the filename list of the symbolic links, rather than the data files themselves. This makes it difficult to determine the data provenance at a later date.

2.8 Large Datasets

Large datasets are those files that are comparable in size to the amount of random access 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 work with a 300 MB files 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 even fail. There is no easy solution for this. The best strategy is to work on a machine with sufficient amounts of memory and swap space. Since about 2004, many users have begun to produce or analyze files exceeding 2 GB in size. These users should familiarize themselves with NCO's Large File Support (LFS) capabilities (see Section 3.10 [Large File Support], page 29). The next section will increase your familiarity with NCO's memory requirements. With this knowledge you may re-design your data reduction approach to divide the problem into pieces solvable in memory-limited situations.

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. On Cray systems, try to increase the available memory with the `ilimit` command. If you get a memory-related core dump (e.g., '`Error exit (core dumped)`') on a GNU/Linux system, try increasing the process-available memory with `ulimit`.

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`' (or '`--debug-level`', or '`--dbg_lvl`') 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 2.9 [Memory Requirements], page 17.

2.9 Memory Requirements

Many people use NCO on gargantuan files which dwarf the memory available (free RAM plus swap space) even on today's powerful machines. These users want NCO to consume the least memory possible so that their scripts do not have to tediously cut files into smaller pieces that fit into memory. We commend these greedy users for pushing NCO to its limits!

This section describes the memory NCO requires during operation. The required memory is based on the underlying algorithms. The description below is the memory usage per thread. Users with shared memory machines may use the threaded NCO operators (see Section 3.3 [OpenMP Threading], page 19). The peak and sustained memory usage will scale accordingly, i.e., by the number of threads. Memory consumption patterns of all operators are similar, with the exception of `ncap2`.

2.9.1 Single and Multi-file Operators

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 operate on one single record (i.e., time-slice) at a time, wherease the ensemble operators retrieve the entire variable into memory. Let MS be the peak sustained memory demand of an operator, FT be the memory required to store the entire contents of all the variables to be processed in an input file, FR be the memory required to store the entire contents of a single record of each 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 $MS = 2FT+VT$. `ncbo` requires $MS <= 3VT$ (both input variables and the output variable). `ncflint` requires $MS <= 3VT$ (both input variables and the output variable). `ncpdq` requires $MS <= 2VT$ (one input variable and the output variable). `ncwa` requires $MS <= 8VT$ (see below). Note that only variables that are processed, e.g., averaged, concatenated, or differenced, contribute to MS. Variables which do not appear in the output file (see Section 3.11 [Subsetting Variables], page 30) are never read and contribute nothing to the memory requirements.

`ncwa` consumes between two and seven times the memory of a variable in order to process it. Peak consumption occurs when storing simultaneously in memory one input variable, one tally array, one input weight, one conformed/working weight, one weight tally, one input mask, one conformed/working mask, and one output variable. When invoked, the weighting and masking features contribute up to three-sevenths and two-sevenths of these requirements apiece. If weights and masks are *not* specified (i.e., no '`-w`' or '`-a`' options) then `ncwa` requirements drop to $MS <= 3VT$ (one input variable, one tally array, and the output variable).

The above memory requirements must be multiplied by the number of threads *thr_nbr* (see Section 3.3 [OpenMP Threading], page 19). If this causes problems then reduce (with '`-t thr_nbr`') the number of threads.

2.9.2 Memory for `ncap2`

`ncap2` has unique memory requirements due its ability to process arbitrarily long scripts of any complexity. All script acceptable to `ncap2` are ultimately processed as a sequence of binary or unary operations. `ncap2` requires $MS <= 2VT$ under most conditions. An exception to this is when left hand casting (see Section 4.1.1 [Left hand casting], page 56) is used to stretch the size of derived variables beyond the size of any input variables. Let VC be the memory required to store the largest variable defined by left hand casting. In this case, $MS <= 2VC$.

`ncap2` scripts are complete dynamic and may be of arbitrary length. A script that contains many thousands of operations, may uncover a slow memory leak even though each single operation consumes little additional memory. Memory leaks are usually identifiable by their memory usage signature. Leaks cause peak memory usage to increase monotonically with time regardless of script complexity. Slow leaks are very difficult to find. Sometimes a `malloc()` (or `new[]`) failure is the only noticeable clue to their existance. If you have good reasons to believe that a memory allocation failure is ultimately due to an NCO memory leak (rather than inadequate RAM on your system), then we would be very interested in receiving a detailed bug report.

2.10 Performance Limitations

1. No data buffering is performed during `nc_get_var` and `nc_put_var` 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. In the late 1990s it was discovered that some random access operations on large files on certain architectures (e.g., UNICOS) were much slower with NCO than with similar operations performed using languages that bypass the netCDF interface (e.g., Yorick). This may be a penalty of unnecessary byte-swapping in the netCDF interface. It is unclear whether such problems exist in present day (2007) netCDF/NCO environments.

3 NCO Features

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 Internationalization

Availability: All operators

NCO support for *internationalization* of textual input and output (e.g., Warning messages) is nascent. We hope to produce foreign language string catalogues in 2004.

3.2 Metadata Optimization

Availability: `ncatted`, `ncks`, `ncrename`
Short options: None
Long options: '`--hdr_pad`', '`--header_pad`'

NCO supports padding headers to improve the speed of future metadata operations. Use the '`--hdr_pad`' and '`--header_pad`' switches to request that *hdr_pad* bytes be inserted into the metadata section of the output file. Future metadata expansions will not incur the performance penalty of copying the entire output file unless the expansion exceeds the amount of header padding exceeded. This can be beneficial when it is known that some metadata will be added at a future date.

This optimization exploits the netCDF library `nc__enddef()` function, which behaves differently with different versions of netCDF. It will improve speed of future metadata expansion with `CLASSIC` and `64bit` netCDF files, but not necessarily with `NETCDF4` files, i.e., those created by the netCDF interface to the HDF5 library (see Section 3.9 [Selecting Output File Format], page 28).

3.3 OpenMP Threading

Availability: `ncbo`, `ncea`, `ncecat`, `ncflint`, `ncpdq`, `ncra`, `ncrcat`, `ncwa`
Short options: '`-t`'
Long options: '`--thr_nbr`', '`--threads`', '`--omp_num_threads`'

NCO supports shared memory parallelism (SMP) when compiled with an OpenMP-enabled compiler. Threads requests and allocations occur in two stages. First, users may request a specific number of threads *thr_nbr* with the '`-t`' switch (or its long option equiva-

lents, '`--thr_nbr`', '`--threads`', and '`--omp_num_threads`'). If not user-specified, OpenMP obtains *thr_nbr* from the `OMP_NUM_THREADS` environment variable, if present, or from the OS, if not.

NCO may modify *thr_nbr* according to its own internal settings before it requests any threads from the system. Certain operators contain hard-code limits to the number of threads they request. We base these limits on our experience and common sense, and to reduce potentially wasteful system usage by inexperienced users. For example, `ncrcat` is extremely I/O-intensive so we restrict *thr_nbr* <= 2 for `ncrcat`. This is based on the notion that the best performance that can be expected from an operator which does no arithmetic is to have one thread reading and one thread writing simultaneously. In the future (perhaps with netCDF4), we hope to demonstrate significant threading improvements with operators like `ncrcat` by performing multiple simultaneous writes.

Compute-intensive operators (`ncwa` and `ncpdq`) are expected to benefit the most from threading. The greatest increases in throughput due to threading will occur on large dataset where each thread performs millions or more floating point operations. Otherwise, the system overhead of setting up threads may outweigh the theoretical speed enhancements due to SMP parallelism. However, we have not yet demonstrated that the SMP parallelism scales well beyone four threads for these operators. Hence we restrict *thr_nbr* <= 4 for all operators. We encourage users to play with these limits (edit file '`nco_omp.c`') and send us their feedback.

Once the initial *thr_nbr* has been modified for any operator-specific limits, NCO requests the system to allocate a team of *thr_nbr* threads for the body of the code. The operating system then decides how many threads to allocate based on this request. Users may keep track of this information by running the operator with *dbg_lvl* > 0.

By default, operators with thread attach one global attribute to any file they create or modify. The `nco_openmp_thread_number` global attribute contains the number of threads the operator used to process the input files. This information helps to verify that the answers with threaded and non-threaded operators are equal to within machine precision. This information is also useful for benchmarking.

3.4 Command Line Options

Availability: All operators

NCO achieves flexibility by using *command line options*. These options are implemented in all traditional UNIX commands as single letter *switches*, e.g., '`ls -l`'. For many years NCO used only single letter option names. In late 2002, we implemented GNU/POSIX extended or long option names for all options. This was done in a backward compatible way such that the full functionality of NCO is still available through the familiar single letter options. In the future, however, some features of NCO may require the use of long options, simply because we have nearly run out of single letter options. More importantly, mnemonics for single letter options are often non-intuitive so that long options provide a more natural way of expressing intent.

Extended options, also called long options, are implemented using the system-supplied '`getopt.h`' header file, if possible. This provides the `getopt_long` function to NCO[1].

The syntax of *short options* (single letter options) is `-key value` (dash-key-space-value). Here, *key* is the single letter option name, e.g., '`-D 2`'.

The syntax of *long options* (multi-letter options) is `--long_name value` (dash-dash-key-space-value), e.g., '`--dbg_lvl 2`' or `--long_name=value` (dash-dash-key-equal-value), e.g., '`--dbg_lvl=2`'. Thus the following are all valid for the '`-D`' (short version) or '`--dbg_lvl`' (long version) command line option.

```
ncks -D 3 in.nc        # Short option
ncks --dbg_lvl=3 in.nc # Long option, preferred form
ncks --dbg_lvl 3 in.nc # Long option, alternate form
```

The last example is preferred for two reasons. First, '`--dbg_lvl`' is more specific and less ambiguous than '`-D`'. The long option form makes scripts more self documenting and less error prone. Often long options are named after the source code variable whose value they carry. Second, the equals sign `=` joins the key (i.e., *long_name*) to the value in an uninterruptible text block. Experience shows that users are less likely to mis-parse commands when restricted to this form.

GNU implements a superset of the POSIX standard which allows any unambiguous truncation of a valid option to be used.

```
ncks -D 3 in.nc        # Short option
ncks --dbg_lvl=3 in.nc # Long option, full form
ncks --dbg=3 in.nc     # Long option, unambiguous truncation
ncks --db=3 in.nc      # Long option, unambiguous truncation
ncks --d=3 in.nc       # Long option, ambiguous truncation
```

The first four examples are equivalent and will work as expected. The final example will exit with an error since `ncks` cannot disambiguate whether '`--d`' is intended as a truncation of '`--dbg_lvl`', of '`--dimension`', or of some other long option.

NCO provides many long options for common switches. For example, the debugging level may be set in all operators with any of the switches '`-D`', '`--debug-level`', or '`--dbg_lvl`'. This flexibility allows users to choose their favorite mnemonic. For some, it will be '`--debug`' (an unambiguous truncation of '`--debug-level`', and other will prefer '`--dbg`'. Interactive users usually prefer the minimal amount of typing, i.e., '`-D`'. We recommend that scripts which are re-usable employ some form of the long options for future maintainability.

This manual generally uses the short option syntax. This is for historical reasons and to conserve space. The remainder of this manual specifies the full *long_name* of each option. Users are expected to pick the unambiguous truncation of each option name that most suits their taste.

---

[1] If a `getopt_long` function cannot be found on the system, NCO will use the `getopt_long` from the `my_getopt` package by Benjamin Sittler bsittler@iname.com. This is BSD-licensed software available from http://www.geocities.com/ResearchTriangle/Node/9405/#my_getopt.

3.5 Specifying Input Files

Availability (`-n`): `ncea`, `ncecat`, `ncra`, `ncrcat`
Availability (`-p`): All operators
Short options: '`-n`', '`-p`'
Long options: '`--nintap`', '`--pth`', '`--path`'

It is important that users be able to specify multiple input files without typing every filename in full, often a tedious task even by graduate student standards. 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 (or their long option equivalents, '`--nintap`' or '`--pth`' and '`--path`', respectively). 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[2]. 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 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

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

'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.

3.6 Specifying Output Files

Availability: All operators
Short options: '-o'
Long options: '--fl_out', '--output'

NCO commands produce no more than one output file, *fl_out*. Traditionally, users specify *fl_out* as the final argument to the operator, following all input file names. This is the *positional argument* method of specifying input and ouput file names. The positional argument method works well in most applications. NCO also supports specifying *fl_out* using the command line switch argument method, '-o *fl_out*'.

Specifying *fl_out* with a switch, rather than as a positional argument, allows *fl_out* to precede input files in the argument list. This is particularly useful with multi-file operators for three reasons. Multi-file operators may be invoked with hundreds (or more) filenames. Visual or automatic location of *fl_out* in such a list is difficult when the only syntactic distinction between input and output files is their position. Second, specification of a long list of input files may be difficult (see Section 2.7 [Large Numbers of Files], page 14). Making the input file list the final argument to an operator facilitates using `xargs` for this purpose. Some alternatives to `xargs` are very ugly and undesirable. Finally, many users are more comfortable specifying output files with '`-o fl_out`' near the beginning of an argument list. Compilers and linkers are usually invoked this way.

3.7 Accessing Remote Files

Availability: All operators
Short options: '`-p`', '`-l`'
Long options: '`--pth`', '`--path`', '`--lcl`', '`--local`'

All NCO operators can retrieve files from remote sites as well as from the local file system. A remote site can be an anonymous FTP server, a machine on which the user has `rcp`, `scp`, or `sftp` privileges, or NCAR's Mass Storage System (MSS), or an OPeNDAP server. Examples of each are given below, following a brief description of the particular access protocol.

To access a file via an anonymous FTP server, supply the remote file's URL. FTP is an intrinsically insecure protocol because it transfers passwords in plain text format. Users should access sites using anonymous FTP when possible. Some FTP servers require a login/password combination for a valid user account. NCO allows these transactions so long as the required information is stored in the '`.netrc`' file. Usually this information is the remote machine name, login, and password, in plain text, separated by those very keywords, e.g.,

```
machine dust.ess.uci.edu login zender password bushlied
```

Eschew using valuable passwords for FTP transactions, since '`.netrc`' passwords are potentially exposed to eavesdropping software[3].

SFTP, i.e., secure FTP, uses SSH-based security protocols that solve the security issues associated with plain FTP. NCO supports SFTP protocol access to files specified with a homebrew syntax of the form

```
sftp://machine.domain.tld:/path/to/filename
```

Note the second colon following the top-level-domain (tld). This syntax is a hybrid between an FTP URL and a standard remote file syntax.

[3] NCO does not implement command line options to specify FTP logins and passwords because copying those data into the `history` global attribute in the output file (done by default) poses an unacceptable security risk.

To access a file using `rcp` or `scp`, specify the Internet address of the remote file. Of course in this case you must have `rcp` or `scp` 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[4], 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 analyze files stored on remote systems.

```
ncks -l . ftp://dust.ess.uci.edu/pub/zender/nco/in.nc
ncks -l . sftp://dust.ess.uci.edu:/home/ftp/pub/zender/nco/in.nc
ncks -l . dust.ess.uci.edu:/home/zender/nco/data/in.nc
ncks -l . /ZENDER/nco/in.nc
ncks -l . mss:/ZENDER/nco/in.nc
ncks -l . http://dust.ess.uci.edu/cgi-bin/dods/nph-dods/dodsdata/in.nc
```

The first example works verbatim if your system is connected to the Internet and is not behind a firewall. The second example works if you have `sftp` access to the machine `dust.ess.uci.edu`. The third example works if you have `rcp` or `scp` access to the machine `dust.ess.uci.edu`. The fourth and fifth examples work on NCAR computers with local access to the `msrcp`, `msread`, or `nrnet` commands. The sixth command works if your local version of NCO is OPeNDAP-enabled (this is fully described in Section 3.7.1 [OPeNDAP], page 26). The above commands can be rewritten using the '`-p` *input-path*' option as follows:

```
ncks -p ftp://dust.ess.uci.edu/pub/zender/nco -l . in.nc
ncks -p sftp://dust.ess.uci.edu:/home/ftp/pub/zender/nco -l . in.nc
ncks -p dust.ess.uci.edu:/home/zender/nco -l . in.nc
ncks -p /ZENDER/nco -l . in.nc
ncks -p mss:/ZENDER/nco -l . in.nc
ncks -p http://dust.ess.uci.edu/cgi-bin/dods/nph-dods/dodsdata \
     -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

[4] 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`.

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 recommended as it is syntactically more clear.

3.7.1 OPeNDAP

The Distributed Oceanographic Data System (DODS) provides useful replacements for common data interface libraries like netCDF. The DODS versions of these libraries implement network transparent access to data via a client-server data access protocol that uses the HTTP protocol for communication. Although DODS-technology originated with oceanography data, it applyies to virtually all scientific data. In recognition of this, the data access protocol underlying DODS (which is what NCO cares about) has been renamed the Open-source Project for a Network Data Access Protocol, OPeNDAP. We use the terms DODS and OPeNDAP interchangeably, and often write OPeNDAP/DODS for now. In the future we will deprecate DODS in favor of DAP or OPeNDAP, as appropriate[5].

NCO may be DAP-enabled by linking NCO to the OPeNDAP libraries. This is described in the OPeNDAP documentation and automagically implemented in NCO build mechanisms[6]. The '`./configure`' mechanism automatically enables NCO as OPeNDAP clients if it can find the required OPeNDAP libraries[7]. in the usual locations. The `$DODS_ROOT` environment variable may be used to override the default OPeNDAP library location at NCO compile-time. Building NCO with '`bld/Makefile`' and the command `make DODS=Y` adds the (non-intuitive) commands to link to the OPeNDAP libraries installed in the `$DODS_ROOT` directory. The file '`doc/opendap.sh`' contains a generic script intended to help users install OPeNDAP before building NCO. The documentation at the OPeNDAP Homepage is voluminous. Check there and on the DODS mail lists. to learn more about the extensive capabilities of OPeNDAP[8].

Once NCO is DAP-enabled the operators are OPeNDAP clients. All OPeNDAP clients have network transparent access to any files controlled by a OPeNDAP server. Simply specify the input file path(s) in URL notation and all NCO operations may be performed on remote files made accessible by a OPeNDAP server. This command tests the basic functionality of OPeNDAP-enabled NCO clients:

---

[5] DODS is being deprecated because it is ambiguous, referring both to a protocol and to a collection of (oceanography) data. It is superceded by two terms. DAP is the discipline-neutral Data Access Protocol at the heart of DODS. The National Virtual Ocean Data System (NVODS) refers to the collection of oceanography data and oceanographic extensions to DAP. In other words, NVODS is implemented with OPeNDAP. OPeNDAP is *also* the open source project which maintains, develops, and promulgates the DAP standard. OPeNDAP and DAP really are interchangeable. Got it yet?

[6] Automagic support for DODS version 3.2.x was deprecated in December, 2003 after NCO version 2.8.4. NCO support for OPeNDAP versions 3.4.x commenced in December, 2003, with NCO version 2.8.5. NCO support for OPeNDAP versions 3.5.x commenced in June, 2005, with NCO version 3.0.1. NCO support for OPeNDAP versions 3.6.x commenced in June, 2006, with NCO version 3.1.3. NCO support for OPeNDAP versions 3.7.x commenced in January, 2007, with NCO version 3.1.9.

[7] The minimal set of libraries required to build NCO as OPeNDAP clients are, in link order, '`libnc-dap.a`', '`libdap.a`', and '`libxml2`' and '`libcurl.a`'.

[8] We are most familiar with the OPeNDAP ability to enable network-transparent data access. OPeNDAP has many other features, including sophisticated hyperslabbing and server-side processing via *constraint expressions*. If you know more about this, please consider writing a section on "OPeNDAP Capabilities of Interest to NCO Users" for incorporation in the *NCO User's Guide*.

```
% ncks -o ~/foo.nc -C -H -v one -l /tmp \
  -p http://dust.ess.uci.edu/cgi-bin/dods/nph-dods/dodsdata in.nc
one = 1
% ncks -H -v one ~/foo.nc
one = 1
```

The `one = 1` outputs confirm (first) that `ncks` correctly retrieved data via the OPeNDAP protocol and (second) that `ncks` created a valid local copy of the subsetted remote file.

The next command is a more advanced example which demonstrates the real power of OPeNDAP-enabled NCO clients. The `ncwa` client requests an equatorial hyperslab from remotely stored NCEP reanalyses data of the year 1969. The NOAA OPeNDAP server (hopefully!) serves these data. The local `ncwa` client then computes and stores (locally) the regional mean surface pressure (in Pa).

```
ncwa -C -a lat,lon,time -d lon,-10.,10. -d lat,-10.,10. -l /tmp -p \
http://www.cdc.noaa.gov/cgi-bin/nph-nc/Datasets/ncep.reanalysis.dailyavgs/surface \
  pres.sfc.1969.nc ~/foo.nc
```

All with one command! The data in this particular input file also happen to be packed (see Section 4.1.4 [Intrinsic functions], page 59), although this is completely transparent to the user since NCO automatically unpacks data before attempting arithmetic.

NCO obtains remote files from the OPeNDAP server (e.g., '`www.cdc.noaa.gov`') rather than the local machine. Input files are first copied to the local machine, then processed. The OPeNDAP server performs data access, hyperslabbing, and transfer to the local machine. This allows the I/O to appear to NCO as if the input files were local. The local machine performs all arithmetic operations. Only the hyperslabbed output data are transferred over the network (to the local machine) for the number-crunching to begin. The advantages of this are obvious if you are examining small parts of large files stored at remote locations.

3.8 Retaining Retrieved Files

Availability: All operators
Short options: '`-R`'
Long options: '`--rtn`', '`--retain`'

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.

Invoking `-R` disables the default printing behavior of `ncks`. This allows `ncks` to retrieve remote files without automatically trying to print them. See Section 4.7 [ncks netCDF Kitchen Sink], page 82, for more details.

Note that the remote retrieval features of NCO can always be used to retrieve *any* file, including non-netCDF files, via `SSH`, anonymous FTP, or `msrcp`. 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 -R -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.jpg`' as a netCDF file, it fails and exits, leaving '`storm.jpg`' in the current directory.

If your NCO is DAP-enabled (see Section 3.7.1 [OPeNDAP], page 26), then you may use NCO to retrieve any files (including netCDF, HDF, etc.) served by an OPeNDAP server to your local machine. For example,

```
ncks -R -l . -p \
http://www.cdc.noaa.gov/cgi-bin/nph-nc/Datasets/ncep.reanalysis.dailyavgs/surface \
  pres.sfc.1969.nc
```

Note that NCO is never the preffered way to transport files from remote machines. For large jobs, that is best handled by FTP, SSH, or `wget`. It may occasionally be useful to use NCO to transfer files when your other preferred methods are not available locally.

3.9 Selecting Output File Format

Availability: `ncap2`, `ncbo`, `ncea`, `ncecat`, `ncflint`, `ncks`, `ncpdq`, `ncra`, `ncrcat`, `ncwa`
Short options: '`-4`'
Long options: '`--4`', '`--64bit`', '`--fl_fmt`', '`--netcdf4`'

All NCO operators support (read and write) all three (or four, depending on how one counts) file formats supported by netCDF4. The operators listed under "Availability" above allow the user to specify the output file format independent of the input file format. These operators allow the user to convert between the various file formats. The operators `ncatted` and `ncrename` always write the output netCDF file in the same format as the input netCDF file.

netCDF supports four types of files: `CLASSIC`, `64BIT`, `NETCDF4`, and `NETCDF4_CLASSIC`, The `CLASSIC` format is the traditional 32-bit offset written by netCDF2 and netCDF3. As of 2005, most netCDF datasets are in `CLASSIC` format. The `64BIT` format was added in Fall, 2004.

The `NETCDF4` format uses HDF5 as the file storage layer. The files are (usually) created, accessed, and manipulated using the traditional netCDF3 API (with numerous extensions). The `NETCDF4_CLASSIC` format refers to netCDF4 files created with the `NC_CLASSIC_MODEL` mask. Such files use HDF5 as the back-end storage format (unlike netCDF3), though they incorporate only netCDF3 features. Hence `NETCDF4_CLASSIC` files are perfectly readable by applications which use only the netCDF3 API and library. NCO must be built with netCDF4 to write files in the new `NETCDF4` and `NETCDF4_CLASSIC` formats, and to read

files in the new NETCDF4 format. Users are advised to use the default CLASSIC format or the NETCDF4_CLASSIC format until netCDF4 is more widespread. Widespread support for NETCDF4 format files is not expected for a few years, 2007–2008, say. If performance or coolness are issues, then use NETCDF4_CLASSIC instead of CLASSIC format files.

All operators write classic CLASSIC (32-bit offset) format files unless told otherwise. Toggling the long option '--64bit' switch (or its *key-value* equivalent '--fl_fmt=64bit') produces the netCDF3 64-bit offset format named 64BIT. NCO must be built with netCDF 3.6 or higher to produce a 64BIT file. Toggling the '-4' switch (or its long option equivalents '--4' or '--netcdf4'), or setting its *key-value* equivalent '--fl_fmt=netcdf4' produces a NETCDF4 file (i.e., HDF). Casual users are advised to use the default (netCDF3) CLASSIC format until netCDF 3.6 and netCDF 4.0 are more widespread.

These examples show how to convert a file from any netCDF format into any other netCDF format (subject to limits of the format):

```
ncks --fl_fmt=classic in.nc foo_3c.nc # netCDF3 classic
ncks --fl_fmt=64bit in.nc foo_364.nc # netCDF3 64bit
ncks --fl_fmt=netcdf4 in.nc foo_4.nc # netCDF4
ncks --fl_fmt=netcdf4_classic in.nc foo_4c.nc # netCDF4 classic
```

To discover whether a netCDF file is a classic (32-bit offset) or newer 64-bit offset netCDF3 format, or is netCDF4 format, examine it with the od (octal dump) command:

```
% od -An -c -N4 foo_3c.nc
   C   D   F 001
% od -An -c -N4 foo_364.nc
   C   D   F 002
% od -An -c -N4 foo_4.nc
 211   H   D   F
% od -An -c -N4 foo_4c.nc
 211   H   D   F
```

Values of 'C D F 001' and 'C D F 002' indicate 32-bit (classic) and 64-bit netCDF3 formats, respectively, while values of '211 H D F' indicate the newer netCDF4 file format. Note that NETCDF4 and NETCDF4_CLASSIC are the same formats. The latter simply causes an application to fail if it attempts to write a NETCDF4 file that cannot be completely read by the netCDF3 library. As of October, 2005, NCO writes no netCDF4-specific data structures and so always succeeds at writing NETCDF4_CLASSIC files.

3.10 Large File Support

Availability: All operators
Short options: none
Long options: none

NCO has Large File Support (LFS), meaning that NCO can write files larger than 2 GB on some 32-bit operating systems with netCDF libraries earlier than version 3.6. If desired,

LFS support must be configured when both netCDF and NCO are installed. netCDF versions 3.6 and higher support 64-bit file addresses as part of the netCDF standard. We recommend that users ignore LFS support which is difficult to configure and is implemented in NCO only to support netCDF versions prior to 3.6. This obviates the need for configuring explicit LFS support in applications (such as NCO) which now support 64-bit files directly through the netCDF interface. See Section 3.9 [Selecting Output File Format], page 28 for instructions on accessing the different file formats, including 64-bit files, supported by the modern netCDF interface.

If you are still interesting in explicit LFS support for netCDF versions prior to 3.6, know that LFS support depends on a complex, interlocking set of operating system[9] and netCDF suppport issues. The netCDF LFS FAQ at `http://my.unidata.ucar.edu/content/software/netcdf/faq-lfs.html` describes the various file size limitations imposed by different versions of the netCDF standard. NCO and netCDF automatically attempt to configure LFS at build time.

3.11 Subsetting Variables

Availability: (`ncap2`), `ncbo`, `ncea`, `ncecat`, `ncflint`, `ncks`, `ncpdq`, `ncra`, `ncrcat`, `ncwa`
Short options: '`-v`', '`-x`'
Long options: '`--variable`', '`--exclude`' or '`--xcl`'

Subsetting variables refers to explicitly specifying variables to be included or excluded from operator actions. 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.

Variables explicitly specified for extraction with '`-v var[,...]`' *must* be present in the input file or an error will result. Variables explicitly specified for *exclusion* with '`-x -v var[,...]`' need not be present in the input file. Remember, if averaging or concatenating large files stresses your systems memory or disk resources, then the easiest solution is often to use the '`-v`' option to retain only the most important variables (see Section 2.9 [Memory Requirements], page 17).

Due to its special capabilities, `ncap2` interprets the '`-v`' switch differently (see Section 4.1 [ncap2 netCDF Arithmetic Processor], page 56). For `ncap2`, the '`-v`' switch takes no arguments and indicates that *only* user-defined variables should be output. `ncap2` neither accepts nor understands the -x switch.

As of NCO 2.8.1 (August, 2003), variable name arguments of the '`-v`' switch may contain *extended regular expressions*. For example, '`-v '^DST'`' selects all variables beginning with the string '`DST`'. Extended regular expressions are defined by the GNU `egrep` command. The meta-characters used to express pattern matching operations are '`^$+?.*[]{}|`'. If the regular expression pattern matches *any* part of a variable name then that variable is

[9] Linux and AIX are known to support LFS.

selected. This capability is called *wildcarding*, and is very useful for sub-setting large data files.

Because of its wide availability, NCO uses the POSIX regular expression library `regex`. Regular expressions of arbitary complexity may be used. Since netCDF variable names are relatively simple constructs, only a few varieties of variable wildcards are likely to be useful. For convenience, we define the most useful pattern matching operators here:

‘`^`’ Matches the beginning of a string

‘`$`’ Matches the end of a string

‘`.`’ Matches any single character

The most useful repetition and combination operators are

‘`?`’ The preceding regular expression is optional and matched at most once

‘`*`’ The preceding regular expression will be matched zero or more times

‘`+`’ The preceding regular expression will be matched one or more times

‘`|`’ The preceding regular expression will be joined to the following regular expression. The resulting regular expression matches any string matching either subexpression.

To illustrate the use of these operators in extracting variables, consider a file with variables `Q`, `Q01`–`Q99`, `Q100`, `QAA`–`QZZ`, `Q_H2O`, `X_H2O`, `Q_CO2`, `X_CO2`.

```
ncks -v 'Q.?' in.nc              # Variables that contain Q
ncks -v '^Q.?' in.nc             # Variables that start with Q
ncks -v '^Q+.?.' in.nc           # Q, Q0--Q9, Q01--Q99, QAA--QZZ, etc.
ncks -v '^Q..' in.nc             # Q01--Q99, QAA--QZZ, etc.
ncks -v '^Q[0-9][0-9]' in.nc     # Q01--Q99, Q100
ncks -v '^Q[[:digit:]]{2}' in.nc # Q01--Q99
ncks -v 'H2O$' in.nc             # Q_H2O, X_H2O
ncks -v 'H2O$|CO2$' in.nc        # Q_H2O, X_H2O, Q_CO2, X_CO2
ncks -v '^Q[0-9][0-9]$' in.nc    # Q01--Q99
ncks -v '^Q[0-6][0-9]|7[0-3]' in.nc # Q01--Q73, Q100
ncks -v '(Q[0-6][0-9]|7[0-3])$' in.nc # Q01--Q73
ncks -v '^[a-z]_[a-z]{3}$' in.nc # Q_H2O, X_H2O, Q_CO2, X_CO2
```

Beware—two of the most frequently used repetition pattern matching operators, ‘`*`’ and ‘`?`’, are also valid pattern matching operators for filename expansion (globbing) at the shell-level. Confusingly, their meanings in extended regular expressions and in shell-level filename expansion are significantly different. In an extended regular expression, ‘`*`’ matches zero or more occurences of the preceding regular expression. Thus ‘`Q*`’ selects all variables, and ‘`Q+.*`’ selects all variables containing ‘`Q`’ (the ‘`+`’ ensures the preceding item matches at least once). To match zero or one occurence of the preceding regular expression, use ‘`?`’. Documentation for the UNIX `egrep` command details the extended regular expressions which NCO supports.

One must be careful to protect any special characters in the regular expression specification from being interpreted (globbed) by the shell. This is accomplish by enclosing special characters within single or double quotes

```
ncra -v Q?? in.nc out.nc   # Error: Shell attempts to glob wildcards
ncra -v '^Q+..' in.nc out.nc # Correct: NCO interprets wildcards
ncra -v '^Q+..' in*.nc out.nc # Correct: NCO interprets, Shell globs
```

The final example shows that commands may use a combination of variable wildcarding and shell filename expansion (globbing). For globbing, '`*`' and '`?`' *have nothing to do* with the preceding regular expression! In shell-level filename expansion, '`*`' matches any string, including the null string and '`?`' matches any single character. Documentation for `bash` and `csh` describe the rules of filename expansion (globbing).

3.12 Subsetting Coordinate Variables

Availability: `ncap2`, `ncbo`, `ncea`, `ncecat`, `ncflint`, `ncks`, `ncpdq`, `ncra`, `ncrcat`, `ncwa`
Short options: '`-C`', '`-c`'
Long options: '`--no-coords`', '`--no-crd`', '`--crd`', '`--coords`'

By default, coordinates variables associated with any variable appearing in the *input-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. Both '`-c`' and '`-C`' honor the CF `coordinates` convention described in Section 3.27 [CF Conventions], page 51.

3.13 C and Fortran Index conventions

Availability: `ncbo`, `ncea`, `ncecat`, `ncflint`, `ncks`, `ncpdq`, `ncra`, `ncrcat`, `ncwa`
Short options: '`-F`'
Long options: '`--fortran`'

The '`-F`' switch changes NCO to read and write with the Fortran index convention. By default, NCO uses C-style (0-based) indices for all I/O. In C, indices count from 0 (rather than 1), and dimensions are ordered from slowest (inner-most) to fastest (outer-most) varying. In Fortran, indices count from 1 (rather than 0), and dimensions are ordered from fastest (inner-most) to slowest (outer-most) varying. Hence C and Fortran data storage conventions represent mathematical transposes of eachother. Note that record variables contain the record dimension as the most slowly varying dimension. See Section 4.8 [ncpdq

netCDF Permute Dimensions Quickly], page 88 for techniques to re-order (including transpose) dimensions and to reverse data storage order.

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 the C indexing convention, then with Fortran indexing convention results in the following output formats:

```
% ncks -v three_dmn_var in.nc
lat[0]=-90 lev[0]=1000 lon[0]=-180 three_dmn_var[0]=0
...
% ncks -F -v three_dmn_var in.nc
lon(1)=0 lev(1)=100 lat(1)=-90 three_dmn_var(1)=0
...
```

3.14 Hyperslabs

Availability: `ncbo`, `ncea`, `ncecat`, `ncflint`, `ncks`, `ncpdq`, `ncra`, `ncrcat`, `ncwa`
Short options: '`-d` *dim*`,[`*min*`][,[`*max*`][,[`*stride*`]]]`'
Long options: '`--dimension` *dim*`,[`*min*`][,[`*max*`][,[`*stride*`]]]`',
'`--dmn` *dim*`,[`*min*`][,[`*max*`][,[`*stride*`]]]`'

A *hyperslab* is a subset of a variable's data. The coordinates of a hyperslab are specified with the `-d` *dim*`,[`*min*`][,[`*max*`][,[`*stride*`]]]` short option (or with the same arguments to the '`--dimension`' or '`--dmn`' long options). At least one hyperslab argument (*min*, *max*, or *stride*) must be present. 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. The separating comma must be present to indicate the omission of one of these arguments. 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. This convention serves only to differentiate coordinate values from dimension indices. It is independent of the type of any netCDF coordinate variables. For a given dimension, the specified limits must both be coordinate values (with decimal points) or dimension indices (no decimal points). The *stride* option, if any, must be a dimension index, not a coordinate value. See Section 3.15 [Stride], page 34, for more information on the *stride* option.

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 3.17 [Wrapped Coordinates], page 37), 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 currently 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 hyperslabbed dimension requires its own `-d` option).

3.15 Stride

Availability: `ncbo`, `ncea`, `ncecat`, `ncflint`, `ncks`, `ncpdq`, `ncra`, `ncrcat`, `ncwa`
Short options: '`-d` *dim*`,[`*min*`][,[`*max*`][,[`*stride*`]]]`'
Long options: '`--dimension` *dim*`,[`*min*`][,[`*max*`][,[`*stride*`]]]`',
'`--dmn` *dim*`,[`*min*`][,[`*max*`][,[`*stride*`]]]`'

All data operators support specifying a *stride* for any and all dimensions at the same time. The *stride* is the spacing between consecutive points in a hyperslab. A *stride* of 1 picks all the elements of the hyperslab, and a *stride* of 2 skips every other element, etc.. `ncks` multislabs support strides, and are more powerful than the regular hyperslabs supported by the other operators (see Section 3.16 [Multislabs], page 35). Using the *stride* option for the record dimension 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 3.13 [C and Fortran Index Conventions], page 32) 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:

```
for idx in 02 14 26 38 50; do # Bourne Shell
  ncks -d time,${idx} 8501_8912.nc foo.${idx}
done
foreach idx (02 14 26 38 50) # C Shell
  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 4.7 [ncks netCDF Kitchen Sink], page 82, for more information on `ncks`.

Applying the *stride* option to the record dimension in `ncra` and `ncrcat` 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
```

3.16 Multislabs

Availability: `ncks`
Short options: '`-d dim,[min][,[max][,[stride]]]`'
Long options: '`--dimension dim,[min][,[max][,[stride]]]`', '`--dmn dim,[min][,[max][,[stride]]]`'

In late 2002, `ncks` added support for specifying a *multislab* for any variable. A multislab is a union of one or more hyperslabs which is specified by chaining together hyperslab commands, i.e., `-d` options (see Section 3.14 [Hyperslabs], page 33). This allows multislabs to overcome some restraints which limit hyperslabs.

A single `-d` option can only specify a contiguous and/or a regularly spaced multi-dimensional data array. Multislabs are constructed from multiple `-d` options and may therefore have non-regularly spaced arrays. For example, suppose it is desired to operate on all longitudes from 10.0 to 20.0 and from 80.0 to 90.0 degrees. The combined range of longitudes is not selectable in a single hyperslab specfication of the form '`-d dimension,min,max`' or '`-d dimension,min,max,stride`' because its elements are irregularly spaced in coordinate space (and presumably in index space too). The multislab specification for obtaining these values is simply the union of the hyperslabs specifications that comprise the multislab, i.e.,

```
ncks -d lon,10.,20. -d lon,80.,90. in.nc out.nc
ncks -d lon,10.,15. -d lon,15.,20. -d lon,80.,90. in.nc out.nc
```

Any number of hyperslabs specifications may be chained together to specify the multislab.

Users may specify redundant ranges of indices in a multislab, e.g.,

```
ncks -d lon,0,4 -d lon,2,9,2 in.nc out.nc
```

This command retrieves the first five longitudes, and then every other longitude value up to the tenth. Elements 0, 2, and 4 are specified by both hyperslab arguments (hence this is redundant) but will count only once if an arithmetic operation is being performed. This example uses index-based (not coordinate-based) multislabs because the *stride* option only supports index-based hyper-slabbing. See Section 3.15 [Stride], page 34, for more information on the *stride* option.

Multislabs are more efficient than the alternative of sequentially performing hyperslab operations and concatenating the results. This is because NCO employs a novel multislab algorithm to minimize the number of I/O operations when retrieving irregularly spaced data from disk. The NCO multislab algorithm retrieves each element from disk once and only once. Thus users may take some shortcuts in specifying multislabs and the algorithm will obtain the intended values. Specifying redundant ranges is not encouraged, but may be useful on occasion and will not result in unintended consequences.

A final example shows the real power of multislabs. Suppose the Q variable contains three dimensional arrays of distinct chemical constituents in no particular order. We are interested in the NOy species in a certain geographic range. Say that NO, NO2, and N2O5 are elements 0, 1, and 5 of the *species* dimension of Q. The multislab specification might look something like

```
ncks -d species,0,1 -d species,5 -d lon,0,4 -d lon,2,9,2 in.nc out.nc
```

Multislabs are powerful because they may be specified for every dimension at the same time. Thus multislabs obsolete the need to execute multiple `ncks` commands to gather the desired range of data. We envision adding multislab support to all arithmetic operators in the future.

Availability: `ncks`
Short options: '`-d dim,[min][,[max][,[stride]]]`'
Long options: '`--dimension dim,[min][,[max][,[stride]]]`', '`--dmn dim,[min][,[max][,[stride]]]`'

3.17 Wrapped Coordinates

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 are 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*. 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 3.14 [Hyperslabs], page 33).

3.18 UDUnits Support

Availability: `ncbo`, `ncea`, `ncecat`, `ncflint`, `ncks`, `ncpdq`, `ncra`, `ncrcat`, `ncwa`
Short options: '`-d dim,[min][,[max][,[stride]]]`'
Long options: '`--dimension dim,[min][,[max][,[stride]]]`', '`--dmn dim,[min][,[max][,[stride]]]`'

There is more than one way to hyperskin a cat. The UDUnits package provides a library which, if present, NCO uses to translate user-specified physical dimensions into the physical dimensions of data stored in netCDF files. Unidata provides UDUnits under the same terms as netCDF, so sites should install both. Compiling NCO with UDUnits support is currently optional but may become required in a future version of NCO.

Two examples suffice to demonstrate the power and convenience of UDUnits support. First, consider extraction of a variable containing non-record coordinates with physical dimensions stored in MKS units. In the following example, the user extracts all wavelengths in the visible portion of the spectrum in terms of the units very frequently used in visible spectroscopy, microns:

```
% ncks -C -H -v wvl -d wvl,"0.4 micron","0.7 micron" in.nc
wvl[0]=5e-07 meter
```

The hyperslab returns the correct values because the *wvl* variable is stored on disk with a length dimension that UDUnits recognizes in the `units` attribute. The automagical algorithm that implements this functionality is worth describing since understanding it helps one avoid some potential pitfalls. First, the user includes the physical units of the hyperslab dimensions she supplies, separated by a simple space from the numerical values of the hyperslab limits. She encloses each coordinate specifications in quotes so that the shell does not break the *value-space-unit* string into separate arguments before passing them to NCO. Double quotes ("*foo*") or single quotes ('*foo*') are equally valid for this purpose. Second, NCO recognizes that units translation is requested because each hyperslab argument contains text characters and non-initial spaces. Third, NCO determines whether the *wvl* is dimensioned with a coordinate variable that has a `units` attribute. In this case, *wvl* itself is a coordinate variable. The value of its `units` attribute is `meter`. Thus *wvl* passes this test so UDUnits conversion is attempted. If the coordinate associated with the variable does not contain a `units` attribute, then NCO aborts. Fourth, NCO passes the specified and desired dimension strings (microns are specified by the user, meters are required by NCO) to the UDUnits library. Fifth, the UDUnits library that these dimension are commensurate and it returns the appropriate linear scaling factors to convert from microns to meters to NCO. If the units are incommensurate (i.e., not expressible in the same fundamental MKS units), or are not listed in the UDUnits database, then NCO aborts since it cannot determine the user's intent. Finally, NCO uses the scaling information to convert the user-specified hyperslab limits into the same physical dimensions as those of the corresponding cooridinate variable on disk. At this point, NCO can perform a coordinate hyperslab using the same algorithm as if the user had specified the hyperslab without requesting units conversion.

The translation and dimensional innterpretation of time coordinates shows a more powerful, and probably more common, UDUnits application. In this example, the user prints all data between the eighth and ninth of December, 1999, from a variable whose time dimension is hours since the year 1900:

```
% ncks -H -C -v time_udunits -d time_udunits,"1999-12-08 \
  12:00:0.0","1999-12-09 00:00:0.0",2 in.nc foo2.nc
time_udunits[1]=876018 hours since 1900-01-01 00:00:0.0
```

Here, the user invokes the stride (see Section 3.15 [Stride], page 34) capability to obtain every other timeslice. This is possible because the UDUnits feature is additive, not exclusive—it works in conjunction with all other hyperslabbing (see Section 3.14 [Hyperslabs], page 33) options and in all operators which support hyperslabbing. The following example shows how one might average data in a time period spread across multiple input files

```
ncra -d time,"1939-09-09 12:00:0.0","1945-05-08 00:00:0.0" \
  in1.nc in2.nc in3.nc out.nc
```

Note that there is no excess whitespace before or after the individual elements of the '`-d`' argument. This is important since, as far as the shell knows, '`-d`' takes only *one* command-line argument. Parsing this argument into its component *`dim`*`,[`*`min`*`][,[`*`max`*`][,[`*`stride`*`]]]` elements (see Section 3.14 [Hyperslabs], page 33) is the job of NCO. When unquoted whitespace is present between these elements, the shell passes NCO arugment fragments which will not parse as intended.

NCO implemented support for the UDUnits2 library with version 3.9.2 (August, 2007). The UDUnits2 package supports non-ASCII characters and logarithmic units. We are interested in user-feedback on these features, which are un-tested with NCO.

The UDUnits package documentation describes the supported formats of time dimensions. Among the metadata conventions which adhere to these formats are the Climate and Forecast (CF) Conventions and the Cooperative Ocean/Atmosphere Research Data Service (COARDS) Conventions. The following '`-d arguments`' extract the same data using commonly encountered time dimension formats:

```
-d time,"1918-11-11 11:00:0.0","1939-09-09 00:00:0.0"
```

All of these formats include at least one dash `-` in a non-leading character position (a dash in a leading character position is a negative sign). NCO assumes that a non-leading dash in a limit string indicates that a UDUnits date conversion is requested.

netCDF variables should always be stored with MKS (i.e., God's) units, so that application programs may assume MKS dimensions apply to all input variables. The UDUnits feature is intended to alleviate some of the NCO user's pain when handling MKS units. It connects users who think in human-friendly units (e.g., miles, millibars, days) to extract data which are always stored in God's units, MKS (e.g., meters, Pascals, seconds). The feature is not intended to encourage writers to store data in esoteric units (e.g., furlongs, pounds per square inch, fortnights).

3.19 Missing values

Availability: `ncap2`, `ncbo`, `ncea`, `ncflint`, `ncpdq`, `ncra`, `ncwa`
Short options: 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 that may contain missing data.

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

   NCO adopted the behavior that the default attribute name, if any, assumed to specify the value of data to ignore is `_FillValue` with version 3.9.2 (August, 2007). Prior to that, the `missing_value` attribute, if any, was assumed to specify the value of data to ignore. Supporting both of these attributes simultaneously is not practical. Hence the behavior NCO once applied to *missing_value* it now applies to any *_FillValue*. NCO now treats any *missing_value* as normal data[10].

   It has been and remains most advisable to create both `_FillValue` and `missing_value` attributes with identical values in datasets. Many legacy datasets contain only `missing_value` attributes. NCO can help migrating datasets between these conventions. One may use `ncrename` (see Section 4.11 [ncrename netCDF Renamer], page 99) to rename all `missing_value` attributes to `_FillValue`:

   ```
   ncrename -a .missing_value,_FillValue inout.nc
   ```

   Alternatively, one may use `ncatted` (see Section 4.2 [ncatted netCDF Attribute Editor], page 65) to add a `_FillValue` attribute to all variables

   ```
   ncatted -O -a _FillValue,,o,f,1.0e36 inout.nc
   ```

2. Converting the *_FillValue* to the type of the variable, if neccessary.

   Consider a variable *var* of type *var_type* with a `_FillValue` attribute of type *att_type* containing the value *_FillValue*. As a guideline, the type of the `_FillValue` 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 *_FillValue* to determine which elements of *var* are to be treated as missing data. If not, then NCO converts *_FillValue* from *att_type* to *var_type* by using the implicit conversion rules of C, or, if *att_type* is `NC_CHAR`[11], by typecasting the results of the C function `strtod(_FillValue)`. You may use the NCO operator `ncatted` to change the `_FillValue` at-

[10] The old functionality, i.e., where the ignored values are indicated by `missing_value` not `_FillValue`, may still be selected *at NCO build time* by compiling NCO with the token definition *`CPPFLAGS='-DNCO_MSS_VAL_SNG=missing_value'`*.

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

tribute and all data whose data is *_FillValue* to a new value (see Section 4.2 [ncatted netCDF Attribute Editor], page 65).

3. Identifying missing data during arithmetic operations.

   When an NCO arithmetic operator processes a variable *var* with a `_FillValue` attribute, it compares each value of *var* to *_FillValue* before performing an operation. Note the *_FillValue* comparison imposes a performance penalty on the operator. Arithmetic processing of variables which contain the `_FillValue` attribute always incurs this penalty, even when none of the data are missing. Conversely, arithmetic processing of variables which do not contain the `_FillValue` attribute never incurs this penalty. In other words, do not attach a `_FillValue` 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 *_FillValue* towards the average. `ncbo` and `ncflint` define a *_FillValue* result when either of the input values is a *_FillValue*. Sometimes the *_FillValue* 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 *_FillValue* assigned to that variable in the current file). Suffice it to say that, in all known cases, NCO does "the right thing".

   It is impossible to determine and store the correct result of a binary operation in a single variable. One such corner case occurs when both operands have differing *_FillValue* attributes, i.e., attributes with different numerical values. Since the output (result) of the operation can only have one *_FillValue*, some information may be lost. In this case, NCO always defines the output variable to have the same *_FillValue* as the first input variable. Prior to performing the arithmetic operation, all values of the second operand equal to the second *_FillValue* are replaced with the first *_FillValue*. Then the arithmetic operation proceeds as normal, comparing each element of each operand to a single *_FillValue*. Comparing each element to two distinct *_FillValue*'s would be much slower and would be no likelier to yield a more satisfactory answer. In practice, judicious choice of *_FillValue* values prevents any important information from being lost.

3.20 Deflation

Availability: `ncap2`, `ncbo`, `ncea`, `ncecat`, `ncflint`, `ncks`, `ncpdq`, `ncra`, `ncrcat`, `ncwa`
Short options: '`-L`'
Long options: '`--dfl_lvl`', '`--deflate`'

All NCO operators that define variables support the netCDF4 feature of storing variables compressed with Lempel-Ziv deflation. Activate this deflation with the `-L` *`dfl_lvl`* short option (or with the same argument to the '`--dfl_lvl`' or '`--deflate`' long options). Deflation uses lossless techniques to compress data. Specify the deflation level *dfl_lvl* on a scale from no deflation (*dfl_lvl = 0*) to maximum deflation (*dfl_lvl = 9*). Higher deflation levels require more time for compression. File sizes resulting from minimal (*dfl_lvl = 1*) and maximal (*dfl_lvl = 9*) deflation levels typically differ by a few percent.

To compress an entire file using deflation, use

```
ncks -4 -L 0 in.nc out.nc # No deflation (fast, no time penalty)
ncks -4 -L 1 in.nc out.nc # Minimal deflation (little time penalty)
ncks -4 -L 9 in.nc out.nc # Maximal deflation (much slower)
```

Unscientific testing shows that deflation compresses typical climate datasets by 30-60%. Packing, a lossy compression technique available for all netCDF files (see Section 3.21 [Packed data], page 42), can easily compress files by 50%. Packed data may be deflated to squeeze datasets by about 80%.

```
ncks  -4 -L 1 in.nc out.nc # Minimal deflation (~30-60% compression)
ncks  -4 -L 9 in.nc out.nc # Maximal deflation (~31-63% compression)
ncpdq         in.nc out.nc # Standard packing  (~50% compression)
ncpdq -4 -L 9 in.nc out.nc # Deflated packing  (~80% compression)
```

3.21 Packed data

Availability: `ncap2`, `ncbo`, `ncea`, `ncflint`, `ncpdq`, `ncra`, `ncwa`
Short options: None

The phrase *packed data* refers to data which are stored in the standard netCDF3 packing format which employs a lossy algorithm. See Section 4.7 [ncks netCDF Kitchen Sink], page 82 for a description of deflation, a lossless compression technique available with netCDF4 only. Packed data may be deflated to save additional space.

Packing Algorithm

Packing The standard netCDF packing algorithm is lossy, and produces data with the same dynamic range as the original but which requires no more than half the space to store. The packed variable is stored (usually) as type `NC_SHORT` with the two attributes required to unpack the variable, `scale_factor` and `add_offset`, stored at the original (unpacked) precision of the variable[12]. Let *min* and *max* be the minimum and maximum values of *x*.

$$\begin{aligned} \text{scale\_factor} &= (\max - \min)/\text{ndrv} \\ \text{add\_offset} &= (\min + \max)/2 \\ \text{pck} &= (\text{upk} - \text{add\_offset})/\text{scale\_factor} \\ &= \frac{\text{ndrv} \times [\text{upk} - (\min + \max)/2]}{\max - \min} \end{aligned}$$

where *ndrv* is the number of discrete representable values for given type of packed variable. The theoretical maximum value for *ndrv* is two raised to the number of bits used to store the packed variable. Thus if the variable is packed into type `NC_SHORT`, a two-byte datatype, then there are at most $2^16 = 65536$ distinct values representible. In practice, the number of discretely representible values is taken to be one less than the theoretical maximum. This

[12] Although not a part of the standard, NCO enforces the policy that the `_FillValue` attribute, if any, of a packed variable is also stored at the original precision.

leaves extra space and solves potential problems with rounding which can occur during the unpacking of the variable. Thus for `NC_SHORT`, $ndrv = 65536 - 1 = 65535$. Less often, the variable may be packed into type `NC_CHAR`, where $ndrv = 256 - 1 = 255$, or type `NC_INT` where where $ndrv = 4294967295 - 1 = 4294967294$. One useful feature of (lossy) netCDF packing algorithm is that additional, loss-less packing algorithms perform well on top of it.

Unpacking Algorithm

Unpacking The unpacking algorithm depends on the presence of two attributes, `scale_factor` and `add_offset`. If `scale_factor` is present for a variable, the data are multiplied by the value *scale_factor* after the data are read. If `add_offset` is present for a variable, then the *add_offset* value is added to the data after the data are read. If both `scale_factor` and `add_offset` attributes are present, the data are first scaled by *scale_factor* before the offset *add_offset* is added.

$$\begin{aligned} \text{upk} &= \text{scale\_factor} \times \text{pck} + \text{add\_offset} \\ &= \frac{\text{pck} \times (\max - \min)}{\text{ndrv}} + \frac{\min + \max}{2} \end{aligned}$$

When `scale_factor` and `add_offset` are used for packing, the associated variable (containing the packed data) is typically of type `byte` or `short`, whereas the unpacked values are intended to be of type `int`, `float`, or `double`. An attribute's `scale_factor` and `add_offset` and `_FillValue`, if any, should all be of the type intended for the unpacked data, i.e., `int`, `float` or `double`.

Default Handling of Packed Data

All NCO arithmetic operators understand packed data. The operators automatically unpack any packed variable in the input file which will be arithmetically processed. For example, `ncra` unpacks all record variables, and `ncwa` unpacks all variable which contain a dimension to be averaged. These variables are stored unpacked in the output file.

On the other hand, arithmetic operators do not unpack non-processed variables. For example, `ncra` leaves all non-record variables packed, and `ncwa` leaves packed all variables lacking an averaged dimension. These variables (called fixed variables) are passed unaltered from the input to the output file. Hence fixed variables which are packed in input files remain packed in output files. Completely packing and unpacking files is easily accomplished with `ncpdq` (see Section 4.8 [ncpdq netCDF Permute Dimensions Quickly], page 88). Packing and unpacking individual variables may be done with `ncpdq` and the `ncap2` `pack()` and `unpack()` functions (see Section 4.1.4 [Intrinsic functions], page 59).

3.22 Operation Types

Availability: `ncap2`, `ncra`, `ncea`, `ncwa`
Short options: '`-y`'
Long options: '`--operation`', '`--op_typ`'

The '`-y` *op_typ*' switch allows specification of many different types of operations Set *op_typ* to the abbreviated key for the corresponding operation:

`avg`	Mean value (default)
`sqravg`	Square of the mean
`avgsqr`	Mean of sum of squares
`max`	Maximium value
`min`	Minimium value
`rms`	Root-mean-square (normalized by *N*)
`rmssdn`	Root-mean square (normalized by *N-1*)
`sqrt`	Square root of the mean
`ttl`	Sum of values

NCO assumes coordinate variables represent grid axes, e.g., longitude. The only rank-reduction which makes sense for coordinate variables is averaging. Hence NCO implements the operation type requested with '`-y`' on all non-coordinate variables, but not on coorniate variables. When an operation requires a coordinate variable to be reduced in rank, i.e., from one dimension to a scalar or from one dimension to a degenerate (single value) array, then NCO *always averages* the coordinate variable regardless of the arithmetic operation type performed on the non-coordinate variables.

The mathematical definition of each arithmetic operation is given below. See Section 4.12 [ncwa netCDF Weighted Averager], page 101, for additional information on masks and normalization. If an operation type is not specified with '`-y`' then the operator performs an arithmetic average by default. Averaging is described first so the terminology for the other operations is familiar.

The masked, weighted average of a variable x can be generally represented as

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

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 too early to note, however, that when $\mu_i = m_i = w_i = 1$, the generic averaging expression above reduces to a simple arithmetic average. Furthermore, $m_i = w_i = 1$ for all operators except `ncwa`. These variables are included in the discussion below for completeness, and for possible future use in other operators.

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 3.19 [Missing Values], page 40) for the variable.

2. x_i is located at a point where the mask condition (see Section 4.12.1 [Mask condition], page 102) 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.

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 3.19 [Missing Values], page 40) for the variable. Thus, for a variable with no `_FillValue` attribute, μ_i is always 1. All NCO arithmetic operators (`ncbo`, `ncra`, `ncea`, `ncflint`, `ncwa`) treat missing values analogously.

Besides (weighted) averaging, `ncwa`, `ncra`, and `ncea` also compute some common non-linear operations which may be specified with the '`-y`' switch (see Section 3.22 [Operation Types], page 43). The other rank-reducing operations are simple variations of the generic weighted mean described above. The total value of x (`-y ttl`) is

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

Note that the total is the same as the numerator of the mean of x, and may also be obtained in `ncwa` by using the '`-N`' switch (see Section 4.12 [ncwa netCDF Weighted Averager], page 101).

The minimum value of x (`-y min`) is

$$\bar{x}_j = \min[\mu_1 m_1 w_1 x_1, \mu_2 m_2 w_2 x_2, \ldots, \mu_N m_N w_N x_N]$$

Analogously, the maximum value of x (`-y max`) is

$$\bar{x}_j = \max[\mu_1 m_1 w_1 x_1, \mu_2 m_2 w_2 x_2, \ldots, \mu_N m_N w_N x_N]$$

Thus the minima and maxima are determined after any weights are applied.

The square of the mean value of x (`-y sqravg`) is

$$\bar{x}_j = \left(\frac{\sum_{i=1}^{i=N} \mu_i m_i w_i x_i}{\sum_{i=1}^{i=N} \mu_i m_i w_i} \right)^2$$

The mean of the sum of squares of x (`-y avgsqr`) is

$$\bar{x}_j = \frac{\sum_{i=1}^{i=N} \mu_i m_i w_i x_i^2}{\sum_{i=1}^{i=N} \mu_i m_i w_i}$$

If x represents a deviation from the mean of another variable, $x_i = y_i - \bar{y}$ (possibly created by `ncbo` in a previous step), then applying `avgsqr` to x computes the approximate variance of y. Computing the true variance of y requires subtracting 1 from the denominator, discussed below. For a large sample size however, the two results will be nearly indistinguishable.

The root mean square of x (`-y rms`) is

$$\bar{x}_j = \sqrt{\frac{\sum_{i=1}^{i=N} \mu_i m_i w_i x_i^2}{\sum_{i=1}^{i=N} \mu_i m_i w_i}}$$

Thus `rms` simply computes the squareroot of the quantity computed by `avgsqr`.

The root mean square of x with standard-deviation-like normalization (`-y rmssdn`) is implemented as follows. When weights are not specified, this function is the same as the root mean square of x except one is subtracted from the sum in the denominator

$$\bar{x}_j = \sqrt{\frac{\sum_{i=1}^{i=N} \mu_i m_i x_i^2}{-1 + \sum_{i=1}^{i=N} \mu_i m_i}}$$

If x represents the deviation from the mean of another variable, $x_i = y_i - \bar{y}$, then applying `rmssdn` to x computes the standard deviation of y. In this case the -1 in the denominator compensates for the degree of freedom already used in computing $\bar{y}$ in the numerator. Consult a statistics book for more details.

When weights are specified it is unclear how to compensate for this extra degree of freedom. Weighting the numerator and denominator of the above by w_i and subtracting one from the denominator is only appropriate when all the weights are 1.0. When the weights are arbitrary (e.g., Gaussian weights), subtracting one from the sum in the denominator does not necessarily remove one degree of freedom. Therefore when `-y rmssdn` is requested and weights are specified, `ncwa` actually implements the `rms` procedure. `ncea` and `ncra`, which do not allow weights to be specified, always implement the `rmssdn` procedure when asked.

The square root of the mean of x (`-y sqrt`) is

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

The definitions of some of these operations are not universally useful. Mostly they were chosen to facilitate standard statistical computations within the NCO framework. We are open to redefining and or adding to the above. If you are interested in having other statistical quantities defined in NCO please contact the NCO project (see Section 1.5 [Help Requests and Bug Reports], page 9).

EXAMPLES

Suppose you wish to examine the variable `prs_sfc(time,lat,lon)` which contains a time series of the surface pressure as a function of latitude and longitude. Find the minimium value of `prs_sfc` over all dimensions:

```
ncwa -y min -v prs_sfc in.nc foo.nc
```

Find the maximum value of `prs_sfc` at each time interval for each latitude:

```
ncwa -y max -v prs_sfc -a lon in.nc foo.nc
```

Find the root-mean-square value of the time-series of `prs_sfc` at every gridpoint:

```
ncra -y rms -v prs_sfc in.nc foo.nc
ncwa -y rms -v prs_sfc -a time in.nc foo.nc
```

The previous two commands give the same answer but `ncra` is preferred because it has a smaller memory footprint. Also, by default, `ncra` leaves the (degenerate) `time` dimension in

the output file (which is usually useful) whereas `ncwa` removes the `time` dimension (unless '`-b`' is given).

These operations work as expected in multi-file operators. Suppose that `prs_sfc` is stored in multiple timesteps per file across multiple files, say '`jan.nc`', '`feb.nc`', '`march.nc`'. We can now find the three month maximium surface pressure at every point.

```
ncea -y max -v prs_sfc jan.nc feb.nc march.nc out.nc
```

It is possible to use a combination of these operations to compute the variance and standard deviation of a field stored in a single file or across multiple files. The procedure to compute the temporal standard deviation of the surface pressure at all points in a single file '`in.nc`' involves three steps.

```
ncwa -O -v prs_sfc -a time in.nc out.nc
ncbo -O -v prs_sfc in.nc out.nc out.nc
ncra -O -y rmssdn out.nc out.nc
```

First construct the temporal mean of `prs_sfc` in the file '`out.nc`'. Next overwrite '`out.nc`' with the anomaly (deviation from the mean). Finally overwrite '`out.nc`' with the root-mean-square of itself. Note the use of '`-y rmssdn`' (rather than '`-y rms`') in the final step. This ensures the standard deviation is correctly normalized by one fewer than the number of time samples. The procedure to compute the variance is identical except for the use of '`-y var`' instead of '`-y rmssdn`' in the final step.

`ncap2` can also compute statistics like standard deviations. Brute-force implementation of formulae is one option, e.g.,

```
ncap2 -s 'prs_sfc_sdn=sqrt((prs_sfc-prs_sfc.avg($time)^2).total($time))/($time.size-1)
      in.nc out.nc
```

The operation may, of course, be broken into multiple steps in order to archive intermediate quantities, such as the time-anomalies

```
ncap2 -s 'prs_sfc_anm=prs_sfc-prs_sfc.avg($time)' \
      -s 'prs_sfc_sdn=sqrt((prs_sfc_anm^2).total($time))/($time.size-1)' \
      in.nc out.nc
```

`ncap2` supports intrinsic standard deviation functions (see Section 3.22 [Operation Types], page 43) which simplify the above expression to

```
ncap2 -s 'prs_sfc_sdn=(prs_sfc-prs_sfc.avg($time)).rmssdn($time)' in.nc out.nc
```

These instrinsic functions compute the answer quickly and concisely.

The procedure to compute the spatial standard deviation of a field in a single file '`in.nc`' involves three steps.

```
ncwa -O -v prs_sfc,gw -a lat,lon -w gw in.nc out.nc
ncbo -O -v prs_sfc,gw in.nc out.nc out.nc
ncwa -O -y rmssdn -v prs_sfc -a lat,lon -w gw out.nc out.nc
```

First the appropriately weighted (with '`-w gw`') spatial mean values are written to the output file. This example includes the use of a weighted variable specified with '`-w gw`'. When using weights to compute standard deviations one must remember to include the

weights in the initial output files so that they may be used again in the final step. The initial output file is then overwritten with the gridpoint deviations from the spatial mean. Finally the root-mean-square of the appropriately weighted spatial deviations is taken.

The `ncap2` solution to the spatially-weighted standard deviation problem is

```
ncap2 -s 'prs_sfc_sdn=(prs_sfc*gw-prs_sfc*gw.avg($lat,$lon)).rmssdn($lat,$lon)' \
      in.nc out.nc
```

Be sure to multiply the variable by the weight prior to computing the the anomalies and the standard deviation.

The procedure to compute the standard deviation of a time-series across multiple files involves one extra step since all the input must first be collected into one file.

```
ncrcat -O -v tpt in.nc in.nc foo1.nc
ncwa -O -a time foo1.nc foo2.nc
ncbo -O -v tpt foo1.nc foo2.nc foo2.nc
ncra -O -y rmssdn foo2.nc out.nc
```

The first step assembles all the data into a single file. This may require a lot of temporary disk space, but is more or less required by the `ncbo` operation in the third step.

3.23 Type Conversion

Availability: `ncap2`, `ncbo`, `ncea`, `ncra`, `ncwa`
Short options: None

Type conversion (often called *promotion* or *demotion*) refers to the casting of one fundamental data type to another, e.g., converting `NC_SHORT` (two bytes) to `NC_DOUBLE` (eight bytes). Type conversion is automatic when the language carries out this promotion according to an internal set of rules without explicit user intervention. In contrast, manual type conversion refers to explicit user commands to change the type of a variable or attribute. Most type conversion happens automatically, yet there are situations in which manual type conversion is advantageous.

3.23.1 Automatic type conversion

As a general rule, automatic type conversions should be avoided for at least two reasons. First, type conversions are expensive since they require creating (temporary) buffers and casting each element of a variable from the type it was stored at to some other type. Second, the dataset's creator probably had a good reason for storing data as, say, `NC_FLOAT` rather than `NC_DOUBLE`. In a scientific framework there is no reason to store data with more precision than the observations were made. Thus NCO tries to avoid performing automatic type conversions when performing arithmetic.

Automatic type conversion during arithmetic in the languages C and Fortran is performed only when necessary. All operands in an operation are converted to the most precise type before the operation takes place. However, following this parsimonious conversion rule dogmatically results in numerous headaches. For example, the average of the two `NC_SHORT`s

17000s and 17000s results in garbage since the intermediate value which holds their sum is also of type NC_SHORT and thus cannot represent values greater than 32,767[13]. There are valid reasons for expecting this operation to succeed and the NCO philosophy is to make operators do what you want, not what is most pure. Thus, unlike C and Fortran, but like many other higher level interpreted languages, NCO arithmetic operators will perform automatic type conversion when all the following conditions are met[14]:

1. The operator is ncea, ncra, or ncwa. ncbo is not yet included in this list because subtraction did not benefit from type conversion. This will change in the future
2. The arithmetic operation could benefit from type conversion. Operations that could benefit (e.g., from larger representable sums) include averaging, summation, or any "hard" arithmetic. Type conversion does not benefit searching for minima and maxima ('-y min', or '-y max').
3. The variable on disk is of type NC_BYTE, NC_CHAR, NC_SHORT, or NC_INT. Type NC_DOUBLE is not type converted because there is no type of higher precision to convert to. Type NC_FLOAT is not type converted because, in our judgement, the performance penalty of always doing so would outweigh the (extremely rare) potential benefits.

When these criteria are all met, the operator promotes the variable in question to type NC_DOUBLE, performs all the arithmetic operations, casts the NC_DOUBLE type back to the original type, and finally writes the result to disk. The result written to disk may not be what you expect, because of incommensurate ranges represented by different types, and because of (lack of) rounding. First, continuing the above example, the average (e.g., '-y avg') of 17000s and 17000s is written to disk as 17000s. The type conversion feature of NCO makes this possible since the arithmetic and intermediate values are stored as NC_DOUBLEs, i.e., 34000.0d and only the final result must be represented as an NC_SHORT. Without the type conversion feature of NCO, the average would have been garbage (albeit predictable garbage near -15768s). Similarly, the total (e.g., '-y ttl') of 17000s and 17000s written to disk is garbage (actually -31536s) since the final result (the true total) of 34000 is outside the range of type NC_SHORT.

Type conversions use the floor function to convert floating point number to integers. Type conversions do not attempt to round floating point numbers to the nearest integer. Thus the average of 1s and 2s is computed in double precisions arithmetic as $(1.0d + 1.5d)/2) = 1.5d$. This result is converted to NC_SHORT and stored on disk as $floor(1.5d) = 1s$[15]. Thus no "rounding up" is performed. The type conversion rules of C can be stated as follows: If n is an integer then any floating point value x satisfying $n \leq x < n + 1$ will have the value n when converted to an integer.

3.23.2 Manual type conversion

ncap2 provides intrinsic functions for performing manual type conversions. This, for example, converts variable tpt to external type NC_SHORT (a C-type short), and variable prs to external type NC_DOUBLE (a C-type double).

[13] $32767 = 2^{15} - 1$

[14] Operators began performing type conversions before arithmetic in NCO version 1.2, August, 2000. Previous versions never performed unnecessary type conversion for arithmetic.

[15] The actual type conversions are handled by intrinsic C-language type conversion, so the floor() function is not explicitly called, though the results would be the same if it were.

```
ncap2 -s 'tpt=short(tpt);prs=double(prs)' in.nc out.nc
```

See Section 4.1 [ncap2 netCDF Arithmetic Processor], page 56, for more details.

3.24 Batch Mode

Availability: All operators
Short options: '`-O`', '`-A`'
Long options: '`--ovr`', '`--overwrite`', '`--apn`', '`--append`'

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, interactive questions reduce productivity when processing large amounts of data. 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.

3.25 History Attribute

Availability: All operators
Short options: '`-h`'
Long options: '`--hst`', '`--history`'

All operators automatically append a `history` global attribute to any file they create or modify. 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. As of May, 2002, NCO is case-insensitive to the spelling of the `history` attribute name. Thus attributes named `History` or `HISTORY` (which are non-standard and not recommended) will be treated as valid history attributes. When more than one global attribute fits the case-insensitive search for "history", the first one found will be used. `history` attribute To avoid information overkill, all operators have an optional switch ('`-h`', '`--hst`', or '`--history`') to override automatically appending the `history` attribute (see Section 4.2 [ncatted netCDF Attribute Editor], page 65). Note that the '`-h`' switch also turns off writing the `nco_input_file_list` attribute for multi-file operators (see Section 3.26 [File List Attributes], page 51).

3.26 File List Attributes

Availability: `ncea`, `ncecat`, `ncra`, `ncrcat`
Short options: '`-H`'
Long options: '`--fl_lst_in`', '`--file_list`'

Many methods of specifying large numbers of input file names pass these names via pipes, encodings, or argument transfer programs (see Section 2.7 [Large Numbers of Files], page 14). When these methods are used, the input file list is not explicitly passed on the command line. This results in a loss of information since the `history` attribute no longer contains the exact command by which the file was created.

NCO solves this dilemma by archiving input file list attributes. When the input file list to a multi-file operator is specified via `stdin`, the operator, by default, attaches two global attributes to any file they create or modify. The `nco_input_file_number` global attribute contains the number of input files, and `nco_input_file_list` contains the file names, specified as standard input to the multi-file operator. This information helps to verify that all input files the user thinks were piped through `stdin` actually arrived. Without the `nco_input_file_list` attribute, the information is lost forever and the "chain of evidence" would be broken.

The '`-H`' switch overrides (turns off) the default behavior of writing the input file list global attributes when input is from `stdin`. The '`-h`' switch does this too, and turns off the `history` attribute as well (see Section 3.25 [History Attribute], page 50). Hence both switches allows space-conscious users to avoid storing what may amount to many thousands of filenames in a metadata attribute.

3.27 CF Conventions

Availability: `ncbo`, `ncea`, `ncecat`, `ncflint`, `ncra`, `ncwa`
Short options: None

NCO recognizes the Climate and Forecast (CF) metadata conventions, and treats such data (often called history tapes), specially. NCO handles older NCAR model datasets, such as CCM and early CCSM datasets, with its CF rules even though the earlier data may not contain an explicit `Conventions` attribute (e.g., '`CF-1.0`'). We refer to all such data collectively as CF data. Skip this section if you never work with CF data.

The CF netCDF conventions are described at http://www.cgd.ucar.edu/cms/eaton/cf-metadata/CF-1.0.html. Most CF netCDF conventions are transparent to NCO[16]. There are no known pitfalls associated with using any NCO operator on files adhering to these conventions[17]. However, to facilitate

[16] 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.

[17] The CF conventions recommend `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

maximum user friendliness, NCO does treat certain variables in some CF files specially. The special functions are not required by the CF netCDF conventions, but experience shows they simplify data analysis.

Currently, NCO determines whether a datafile is a CF output datafile simply by checking whether value of the global attribute `Conventions` (if it exists) equals '`CF-1.0`' or '`NCAR-CSM`'. Should `Conventions` equal either of these in the (first) *input-file*, NCO will attempt to treat certain variables specially, because of their meaning in CF files. NCO will not average the following variables often found in CF files: `ntrm`, `ntrn`, `ntrk`, `ndbase`, `nsbase`, `nbdate`, `nbsec`, `mdt`, `mhisf`. These variables contain scalar metadata such as the resolution of the host geophysical model and it makes no sense to change their values.

Furthermore, the `ncbo` operator does not operate on (i.e., add, subtract, etc.) the following variables: `ORO`, `area`, `datesec`, `date`, `gw`, `hyai`, `hyam`, `hybi`. `hybm`, `lat_bnds`, `lon_bnds`, `msk_*`. These variables represent the Gaussian weights, the orography field, time fields, hybrid pressure coefficients, and latititude/longitude boundaries. We call these fields non-coordinate *grid properties*. Coordinate grid properties are easy to identify because they are coordinate variables such as `latitude` and `longitude`.

Users usually want *all* grid properties to remain unaltered in the output file. To be treated as a grid property, the variable name must *exactly* match a name in the above list, or be a coordinate variable. The handling of `msk_*` is exceptional in that *any* variable name beginning with the string `msk_` is considered to be a "mask" and is thus preserved (not operated on arithmetically).

You must spoof NCO if you would like any grid properties or other special CF fields processed normally. For example rename the variables first with `ncrename`, or alter the `Conventions` attribute.

NCO supports the CF `coordinates` convention described at http://www.cgd.ucar.edu/cms/eaton/cf-metadata/CF-1.0.html#grid_ex2. This convention allows variables to specify additional coordinates (including multidimensional coordinates) in a space-separated string attribute named `coordinates`. NCO attaches any such coordinates to the extraction list along with variable and its usual (one-dimensional) coordinates, if any. These auxiliary coordinates are subject to the user-specified overrides described in Section 3.12 [Subsetting Coordinate Variables], page 32.

3.28 ARM Conventions

Availability: `ncrcat`
Short options: None

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 3.28 [ARM Conventions], page 52) and could be fixed for CF files if there is sufficient lobbying.

`ncrcat` has been programmed to correctly handle data files which utilize the Atmospheric Radiation Measurement (ARM) Program convention for time and time offsets. 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 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 conventions $time = base\_time + time\_offset$. 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.

3.29 Operator Version

Availability: All operators
Short options: '`-r`'
Long options: '`--revision`', '`--version`', or '`--vrs`'

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., `1.2`. However using '`-r`' on, say, `ncks`, will produce something like '`NCO netCDF Operators version 1.2 Copyright (C) 1995--2000 Charlie Zender ncks version 1.30 (2000/07/31) "Bolivia"`'. This tells you `ncks` contains all patches up to version `1.30`, which dates from July 31, 2000.

4 Operator Reference Manual

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 19, 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 ncap2 netCDF Arithmetic Processor

SYNTAX

```
ncap2 [-4] [-A] [-C] [-c] [-D dbg] [-F] [-f] [-L dfl_lvl]
[-l path] [-O] [-o output-file] [-p path] [-R] [-r]
[-s algebra] [-S fl.nco] [-v]
input-file [output-file]
```

DESCRIPTION

ncap and ncap2 arithmetically process netCDF files[1]. The processing instructions are contained either in the NCO script file 'fl.nco' or in a sequence of command line arguments. The options '-s' (or long options '--spt' or '--script') are used for in-line scripts and '-S' (or long options '--fl_spt' or '--script-file') are used to provide the filename where (usually multiple) scripting commands are pre-stored. ncap2 was written to perform arbitrary albebraic transformations of data and archive the results as easily as possible. See Section 3.19 [Missing Values], page 40, for treatment of missing values. The results of the algebraic manipulations are called *derived fields*.

Unlike the other operators, ncap2 does not accept a list of variables to be operated on as an argument to '-v' (see Section 3.11 [Subsetting Variables], page 30). Rather, the '-v' switch takes no arguments and indicates that ncap2 should output *only* user-defined variables. ncap2 does not accept or understand the -x switch.

Defining new variables in terms of existing variables is one of ncap2's most powerful features. Derived fields inherit the metadata (i.e., attributes) of their identically named ancestors, if any, in the script or input file. When the derived field is completely new (no identically-named ancestors exist), then it inherits the metadata (if any) of the left-most variable on the right hand side of the defining expression. This metadata inheritance is called *attribute propagation*. Attribute propagation is intended to facilitate well-documented data analysis, and we welcome suggestions to improve this feature.

4.1.1 Left hand casting

The following examples demonstrate the utility of the *left hand casting* ability of ncap2. Consider first this simple, artificial, example. If *lat* and *lon* are one dimensional coordinates of dimensions *lat* and *lon*, respectively, then addition of these two one-dimensional arrays is intrinsically ill-defined because whether *lat_lon* should be dimensioned *lat* by *lon* or *lon* by *lat* is ambiguous (assuming that addition is to remain a *commutative* procedure, i.e., one that does not depend on the order of its arguments). Differing dimensions are said to be *orthogonal* to one another, and sets of dimensions which are mutually exclusive are orthogonal as a set and any arithmetic operation between variables in orthogonal dimensional spaces is ambiguous without further information.

The ambiguity may be resolved by enumerating the desired dimension ordering of the output expression inside square brackets on the left hand side (LHS) of the equals sign. This

[1] ncap2 is the successor to ncap which was put into maintenance mode in November, 2006. This documentation refers to ncap2, which has a superset of the ncap functionality. Eventually ncap will be deprecated in favor ncap2. ncap2 will be renamed ncap in 2007.

is called *left hand casting* because the user resolves the dimensional ordering of the RHS of the expression by specifying the desired ordering on the LHS.

```
ncap2 -s 'lat_lon[lat,lon]=lat+lon' in.nc out.nc
ncap2 -s 'lon_lat[lon,lat]=lat+lon' in.nc out.nc
```

The explicit list of dimensions on the LHS, `[lat,lon]` resolves the otherwise ambiguous ordering of dimensions in *lat_lon*. In effect, the LHS *casts* its rank properties onto the RHS. Without LHS casting, the dimensional ordering of *lat_lon* would be undefined and, hopefully, `ncap2` would print an error message.

Consider now a slightly more complex example. In geophysical models, a coordinate system based on a blend of terrain-following and density-following surfaces is called a *hybrid coordinate system*. In this coordinate system, four variables must be manipulated to obtain the pressure of the vertical coordinate: *PO* is the domain-mean surface pressure offset (a scalar), *PS* is the local (time-varying) surface pressure (usually two horizontal spatial dimensions, i.e, latitude by longitude), *hyam* is the weight given to surfaces of constant density (one spatial dimension, pressure, which is orthogonal to the horizontal dimensions), and *hybm* is the weight given to surfaces of constant elevation (also one spatial dimension). This command constructs a four-dimensional pressure `prs_mdp` from the four input variables of mixed rank and orthogonality:

```
ncap2 -s 'prs_mdp[time,lat,lon,lev]=P0*hyam+PS*hybm' in.nc out.nc
```

Manipulating the four fields which define the pressure in a hybrid coordinate system is easy with left hand casting.

4.1.2 Syntax of `ncap2` statements

Mastering `ncap2` is relatively simple. Each valid statement *statement* consists of standard forward algebraic expression. The '`fl.nco`', if present, is simply a list of such statements, whitespace, and comments. The syntax of statements is most like the computer language C. The following characteristics of C are preserved:

Array syntax
: Arrays elements are placed within `[]` characters;

Array indexing
: Arrays are 0-based;

Array storage
: Last dimension is most rapidly varying;

Assignment statements
: A semi-colon '`;`' indicates the end of an assignment statement.

Comments
: Multi-line comments are enclosed within `/* */` characters. Single line comments are preceded by `//` characters.

Nesting
: Files may be nested in scripts using `#include` *`script`*. Note that the `#include` command is not followed by a semi-colon because it is a pre-processor directive, not an assignment statement. The filename '`script`' is interpreted relative to the run directory.

Attribute syntax
: The at-sign `@` is used to delineate an attribute name from a variable name.

4.1.3 Irregular Grids

NCO is capable of analyzing datasets for many different underlying coordinate grid types. netCDF was developed for and initially used with grids comprised of orthogonal dimensions forming a rectangular coordinate system. We call such grids *standard* grids. It is increasingly common for datasets to use metadata to describe much more complex grids. Let us first define three important coordinate grid properties: rectangularity, regularity, and fxm.

Grids are *regular* if the spacing between adjacent is constant. For example, a 4-by-5 degree latitude-longitude grid is regular because the spacings between adjacent latitudes (4 degrees) are constant as are the (5 degrees) spacings between adjacent longitudes. Spacing in *irregular* grids depends on the location along the coordinate. Grids such as Gaussian grids have uneven spacing in latitude (points cluster near the equator) and so are irregular.

Grids are *rectangular* if the number of elements in any dimension is not a function of any other dimension. For example, a T42 Gaussian latitude-longitude grid is rectangular because there are the same number of longitudes (128) for each of the (64) latitudes. Grids are *non-rectangular* if the elements in any dimension depend on another dimension. Non-rectangular grids present many special challenges to analysis software like NCO.

Wrapped coordinates (see Section 3.17 [Wrapped Coordinates], page 37), such as longitude, are independent of these grid properties (regularity, rectangularity).

The preferred NCO technique to analyze data on non-standard coordinate grids is to create a region mask with `ncap2`, and then to use the mask within `ncap2` for variable-specific processing, and/or with other operators (e.g., `ncwa`, `ncdiff`) for entire file processing.

Before describing the construction of masks, let us review how irregularly gridded geoscience data are described. Say that latitude and longitude are stored as R-dimensional arrays and the product of the dimension sizes is the total number of elements N in the other variables. Geoscience applications tend to use $R = 1$, $R = 2$, and $R = 3$.

If the grid is has no simple representation (e.g., discontinuous) then it makes sense to store all coordinates as 1D arrays with the same size as the number of grid points. These gridpoints can be completely independent of all the other (own weight, area, etc.).

R=1: lat(number_of_gridpoints) and lon(number_of_gridpoints)

If the horizontal grid is time-invariant then R=2 is common:

R=2: lat(south_north,east_west) and lon(south_north,east_west)

The WRF (Weather and Research Forecast) model uses R=3

R=3: lat(time,south_north,east_west), lon(time,south_north,east_west)

and so supports grids that change with time.

Grids with R > 1 often use missing values to indicated empty points. For example, so-called "staggered grids" will use fewer east_west points near the poles and more near the equator. netCDF only accepts rectangular arrays so space must be allocated for the

maximum number of east_west points at all latitudes. Then the application writes missing values into the unused points near the poles.

Let's demonstrate the recommended ncap2 analysis technique by constructing a region mask for an R=2 grid. We wish to find, say, the mean temperature within [lat_min,lat_max] and [lon_min,lon_max]:

```
ncap2 -s 'mask= (lat >= lat_min && lat <= lat_max) && \
                (lon >= lon_min && lon <= lon_max);' in.nc out.nc
```

Once you have a mask, you can use it on specific variables:

```
ncap2 -s 'temperature_avg=(temperature*mask).avg()' in.nc out.nc
```

and you can apply it to entire files:

```
ncwa -a lat,lon -m mask -w area in.nc out.nc
```

You can put this altogether on the command line or in a script, e.g., cleaner.

```
cat > ncap2.in << EOF
mask = (lat >= lat_min && lat <= lat_max) && (lon >= lon_min && > lon <= lon_max);
if(mask.total() > 0){ // Check that mask contains some valid values
  temperature_avg=(temperature*mask).avg(); // Average temperature
  temperature_max=(temperature*mask).max(); // Maximum temperature
}
EOF
ncap2 -S ncap2.in in.nc out.nc
```

For the WRF file creating the mask looks like

```
mask = (XLAT >= lat_min && XLAT <= lat_max) && (XLONG >= lon_min && > XLONG <= lon_max
```

In practice with WRF it's a bit more complicated because you must use the global metadata to determine the grid staggering and offsets to translate XLAT and XLONG into real latitudes and longitudes and missing points. The WRF grid documentation should describe this.

A few notes: Irregular regions are the union of arrays lat/lon_min/max's. The mask procedure is identical for all R.

4.1.4 Intrinsic functions

ncap2 contains a small (and growing) library of intrinsic functions. In addition to the standard mathematical functions (see Section 4.1.6 [Intrinsic mathematical functions], page 60), ncap2 currently supports packing and unpacking.

`pack(x)` The standard packing algorithm is applied to variable x.

`unpack(x)`
The standard unpacking algorithm is applied to variable x.

Type Conversion Functions

These intrinsic functions allow ncap2 to convert variables on disk among the available types supported by netCDF.

`byte(x)` *Convert to* `NC_BYTE` Converts x to external type `NC_BYTE`, a C-type `signed char`.

`char(x)` *Convert to* `NC_CHAR` Converts x to external type `NC_CHAR`, a C-type `unsigned char`.

`double(x)`
Convert to `NC_DOUBLE` Converts x to external type `NC_DOUBLE`, a C-type `double`.

`float(x)` *Convert to* `NC_FLOAT` Converts x to external type `NC_FLOAT`, a C-type `float`.

`int(x)` *Convert to* `NC_INT` Converts x to external type `NC_INT`, a C-type `int`.

`short(x)` *Convert to* `NC_SHORT` Converts x to external type `NC_SHORT`, a C-type `short`.

See Section 3.23 [Type Conversion], page 48, for more details on automatic and manual type conversion.

4.1.6 Intrinsic mathematical functions

`ncap2` supports the standard mathematical functions supplied with most operating systems. Standard calculator notation is used for addition `+`, subtraction `-`, multiplication `*`, division `/`, exponentiation `^`, and modulus `%`. The available elementary mathematical functions are:

`abs(x)` *Absolute value* Absolute value of x, $|x|$. Example: abs$(-1) = 1$

`acos(x)` *Arc-cosine* Arc-cosine of x where x is specified in radians. Example: acos$(1.0) =$ 0.0

`acosh(x)` *Hyperbolic arc-cosine* Hyperbolic arc-cosine of x where x is specified in radians. Example: acosh$(1.0) = 0.0$

`asin(x)` *Arc-sine* Arc-sine of x where x is specified in radians. Example: asin$(1.0) =$ 1.57079632679489661922

`asinh(x)` *Hyperbolic arc-sine* Hyperbolic arc-sine of x where x is specified in radians. Example: asinh$(1.0) = 0.88137358702$

`atan(x)` *Arc-tangent* Arc-tangent of x where x is specified in radians between $-\pi/2$ and $\pi/2$. Example: atan$(1.0) = 0.78539816339744830961$

`atanh(x)` *Hyperbolic arc-tangent* Hyperbolic arc-tangent of x where x is specified in radians between $-\pi/2$ and $\pi/2$. Example: atanh$(3.14159265358979323844) = 1.0$

`ceil(x)` *Ceil* Ceiling of x. Smallest integral value not less than argument. Example: ceil$(0.1) = 1.0$

`cos(x)` *Cosine* Cosine of x where x is specified in radians. Example: cos$(0.0) = 1.0$

`cosh(x)` *Hyperbolic cosine* Hyperbolic cosine of x where x is specified in radians. Example: cosh$(0.0) = 1.0$

`erf(x)` *Error function* Error function of x where x is specified between -1 and 1. Example: erf$(1.0) = 0.842701$

`erfc(x)` *Complementary error function* Complementary error function of x where x is specified between -1 and 1. Example: erfc$(1.0) = 0.15729920705$

`exp(x)` *Exponential* Exponential of x, e^x. Example: $\exp(1.0) = 2.71828182845904523536$

`floor(x)` *Floor* Floor of x. Largest integral value not greater than argument. Example: $\text{floor}(1.9) = 1$

`gamma(x)` *Gamma function* Gamma function of x, $\Gamma(x)$. The well-known and loved continuous factorial function. Example: $\text{gamma}(0.5) = \sqrt{\pi}$

`ln(x)` *Natural Logarithm* Natural logarithm of x, $\ln(x)$. Example: $\ln(2.71828182845904523536) = 1.0$

`log(x)` *Natural Logarithm* Exact synonym for `ln(x)`.

`log10(x)` *Base 10 Logarithm* Base 10 logarithm of x, $\log_{10}(x)$. Example: $\log(10.0) = 1.0$

`nearbyint(x)`
Round inexactly Nearest integer to x is returned in floating point format. No exceptions are raised for *inexact conversions*. Example: $\text{nearbyint}(0.1) = 0.0$

`pow(x,y)` *Power* Value of x is raised to the power of y. Exceptions are raised for *domain errors*. Due to type-limitations in the C language `pow` function, integer arguments are promoted (see Section 3.23 [Type Conversion], page 48) to type `NC_FLOAT` before evaluation. Example: $\text{pow}(2, 3) = 8$

`rint(x)` *Round exactly* Nearest integer to x is returned in floating point format. Exceptions are raised for *inexact conversions*. Example: $\text{rint}(0.1) = 0$

`round(x)` *Round* Nearest integer to x is returned in floating point format. Round halfway cases away from zero, regardless of current IEEE rounding direction. Example: $\text{round}(0.5) = 1.0$

`sin(x)` *Sine* Sine of x where x is specified in radians. Example: $\sin(1.57079632679489661922) = 1.0$

`sinh(x)` *Hyperbolic sine* Hyperbolic sine of x where x is specified in radians. Example: $\sinh(1.0) = 1.1752$

`sqrt(x)` *Square Root* Square Root of x, $\sqrt{x}$. Example: $\text{sqrt}(4.0) = 2.0$

`tan(x)` *Tangent* Tangent of x where x is specified in radians. Example: $\tan(0.78539816339744830961) = 1.0$

`tanh(x)` *Hyperbolic tangent* Hyperbolic tangent of x where x is specified in radians. Example: $\tanh(1.0) = 0.761594155956$

`trunc(x)` *Truncate* Nearest integer to x is returned in floating point format. Round halfway cases toward zero, regardless of current IEEE rounding direction. Example: $\text{trunc}(0.5) = 0.0$

The complete list of mathematical functions supported is platform-specific. Functions mandated by ANSI C are *guaranteed* to be present and are indicated with an asterisk [2]. and

[2] ANSI C compilers are guaranteed to support double precision versions of these functions. These functions normally operate on netCDF variables of type `NC_DOUBLE` without having to perform intrinsic conversions. For example, ANSI compilers provide `sin` for the sine of C-type `double` variables. The

are indicated with an asterisk. Use the '`-f`' (or '`fnc_tbl`' or '`prn_fnc_tbl`') switch to print a complete list of functions supported on your platform. This prints a list of functions and whether they are supported for netCDF variables of intrinsic type `NC_FLOAT` and `NC_DOUBLE`. [3]

EXAMPLES

See the '`ncap.in`' and '`ncap2.in`' scripts released with NCO for more complete demonstrations of `ncap` and `ncap2` functionality, respectively (these scripts are available on-line at http://nco.sf.net/ncap.in and http://nco.sf.net/ncap2.in).

Define new attribute *new* for existing variable *one* as twice the existing attribute *double_att* of variable *att_var*:

```
ncap2 -s 'one@new=2*att_var@double_att' in.nc out.nc
```

Average variables of mixed types (result is of type `double`):

```
ncap2 -s 'average=(var_float+var_double+var_int)/3' in.nc out.nc
```

Multiple commands may be given to `ncap2` in three ways. First, the commands may be placed in a script which is executed, e.g., '`tst.nco`'. Second, the commands may be individually specified with multiple '`-s`' arguments to the same `ncap2` invocation. Third, the commands may be chained together into a single '`-s`' argument to `ncap2`. Assuming the file '`tst.nco`' contains the commands `a=3;b=4;c=sqrt(a^2+b^2);`, then the following `ncap2` invocations produce identical results:

```
ncap2 -v -S tst.nco in.nc out.nc
ncap2 -v -s 'a=3' -s 'b=4' -s 'c=sqrt(a^2+b^2)' in.nc out.nc
ncap2 -v -s 'a=3;b=4;c=sqrt(a^2+b^2)' in.nc out.nc
```

The second and third examples show that `ncap2` does not require that a trailing semicolon '`;`' be placed at the end of a '`-s`' argument, although a trailing semi-colon '`;`' is always allowed. However, semi-colons are required to separate individual assignment statements chained together as a single '`-s`' argument.

`ncap2` may be used to "grow" dimensions, i.e., to increase dimension sizes without altering existing data. Say '`in.nc`' has `ORO(lat,lon)` and the user wishes a new file with `new_ORO(new_lat,new_lon)` that contains zeros in the undefined portions of the new grid.

```
defdim("new_lat",$lat.size+1); // Define new dimension sizes
defdim("new_lon",$lon.size+1);
new_ORO[$new_lat,$new_lon]=0.0f; // Initialize to zero
new_ORO(0:$lat.size-1,0:$lon.size-1)=ORO; // Fill valid data
```

ANSI standard does not require, but many compilers provide, an extended set of mathematical functions that apply to single (`float`) and quadruple (`long double`) precision variables. Using these functions (e.g., `sinf` for `float`, `sinl` for `long double`), when available, is (presumably) more efficient than casting variables to type `double`, performing the operation, and then re-casting. NCO uses the faster intrinsic functions when they are available, and uses the casting method when they are not.

[3] Linux supports more of these intrinsic functions than other OSs.

The commands to define new coordinate variables `new_lat` and `new_lon` in the output file follow a similar pattern. One would might store these commands in a script '`grow.nco`' and then execute the script with

```
ncap2 -v -S grow.nco in.nc out.nc
```

Imagine you wish to create a binary flag based on the value of an array. The flag should have value 1.0 where the array exceeds 1.0, and value 0.0 elsewhere. This example creates the binary flag `ORO_flg` in '`out.nc`' from the continuous array named `ORO` in '`in.nc`'.

```
ncap2 -s 'ORO_flg=(ORO > 1.0)' in.nc out.nc
```

Suppose your task is to change all values of `ORO` which equal 2.0 to the new value 3.0:

```
ncap2 -s 'ORO_msk=(ORO==2.0);ORO=ORO_msk*3.0+!ORO_msk*ORO' in.nc out.nc
```

This creates and uses `ORO_msk` to mask the subsequent arithmetic operation. Values of `ORO` are only changed where `ORO_msk` is true, i.e., where `ORO` equals 2.0. In the future, `ncap2` will support the Fortran90 `where` construct to further simplify this syntax.

This example uses `ncap2` to compute the covariance of two variables. Let the variables u and v be the horizontal wind components. The *covariance* of u and v is defined as the time mean product of the deviations of u and v from their respective time means. Symbolically, the covariance $[u'v'] = [uv] - [u][v]$ where $[x]$ denotes the time-average of x, $[x] \equiv \frac{1}{\tau}\int_{t=0}^{t=\tau} x(t)\,\mathrm{d}t$ and x' denotes the deviation from the time-mean. The covariance tells us how much of the correlation of two signals arises from the signal fluctuations versus the mean signals. Sometimes this is called the *eddy covariance*. We will store the covariance in the variable `uprmvprm`.

```
ncwa -O -a time -v u,v in.nc foo.nc # Compute time mean of u,v
ncrename -O -v u,uavg -v v,vavg foo.nc # Rename to avoid conflict
ncks -A -v uavg,vavg foo.nc in.nc # Place time means with originals
ncap2 -O -s 'uprmvprm=u*v-uavg*vavg' in.nc in.nc # Covariance
ncra -O -v uprmvprm in.nc foo.nc # Time-mean covariance
```

The mathmatically inclined will note that the same covariance would be obtained by replacing the step involving `ncap2` with

```
ncap2 -O -s 'uprmvprm=(u-uavg)*(v-vavg)' foo.nc foo.nc # Covariance
```

As of NCO version 3.1.8 (December, 2006), `ncap2` can compute averages, and thus covariances, by itself:

```
ncap2 -s 'uavg=u.avg($time);vavg=v.avg($time);uprmvprm=u*v-uavg*vavg' \
      -s 'uprmvrpmavg=uprmvprm.avg($time)' in.nc foo.nc
```

We have not seen a simpler method to script and execute powerful arithmetic than `ncap2`.

`ncap2` utilizes many meta-characters (e.g., '`$`', '`?`', '`;`', '`()`', '`[]`') that can confuse the command-line shell if not quoted properly. The issues are the same as those which arise in utilizing extended regular expressions to subset variables (see Section 3.11 [Subsetting Variables], page 30). The example above will fail with no quotes and with double quotes. This

is because shell globbing tries to *interpolate* the value of `$time` from the shell environment unless it is quoted:

```
ncap2 -s 'uavg=u.avg($time)'  in.nc foo.nc # Correct (recommended)
ncap2 -s  uavg=u.avg('$time') in.nc foo.nc # Correct (and dangerous)
ncap2 -s  uavg=u.avg($time)   in.nc foo.nc # Fails ($time = '')
ncap2 -s "uavg=u.avg($time)"  in.nc foo.nc # Fails ($time = '')
```

Without the single quotes, the shell replaces `$time` with an empty string. The command `ncap2` receives from the shell is `uavg=u.avg()`. This causes `ncap2` to average over all dimensions rather than just the *time* dimension, and unintended consequence.

We recommend using single quotes to protect `ncap2` command-line scripts from the shell, even when such protection is not strictly necessary. Expert users may violate this rule to exploit the ability to use shell variables in `ncap2` command-line scripts (see Chapter 6 [CCSM Example], page 109). In such cases it may be necessary to use the shell backslash character '\' to protect the `ncap2` meta-character.

Whether a degenerate record dimension is desirable or undesirable depends on the application. Often a degenerate *time* dimension is useful, e.g., for concatentating, but it may cause problems with arithmetic. Such is the case in the above example, where the first step employs `ncwa` rather than `ncra` for the time-averaging. Of course the numerical results are the same with both operators. The difference is that, unless '`-b`' is specified, `ncwa` writes no *time* dimension to the output file, while `ncra` defaults to keeping *time* as a degenerate (size 1) dimension. Appending `u` and `v` to the output file would cause `ncks` to try to expand the degenerate time axis of `uavg` and `vavg` to the size of the non-degenerate *time* dimension in the input file. Thus the append (`ncks -A`) command would be undefined (and should fail) in this case. Equally important is the '`-C`' argument (see Section 3.12 [Subsetting Coordinate Variables], page 32) to `ncwa` to prevent any scalar *time* variable from being written to the output file. Knowing when to use `ncwa -a time` rather than the default `ncra` for time-averaging takes, well, time.

4.2 `ncatted` netCDF Attribute Editor

SYNTAX

```
ncatted [-a att_dsc] [-a ...] [-D dbg] [-h] [--hdr_pad nbr]
[-l path] [-O] [-o output-file] [-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. This saves time when changing attribute conventions throughout a file. Note that `ncatted` interprets character attributes (i.e., attributes of type `NC_CHAR`) as strings.

Because repeated use of `ncatted` can considerably increase the size of the `history` global attribute (see Section 3.25 [History Attribute], page 50), 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 `_FillValue` 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 3.19 [Missing Values], page 40, for more information.

The key to mastering `ncatted` is understanding the meaning of the structure describing the attribute modification, *att_dsc* specified by the required option '`-a`' or '`--attribute`'. 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. If *att_nm* is omitted (i.e., left blank) and *Delete* mode is selected, then all attributes associated with the specified variable will be deleted.

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. If *att_nm* is omitted (left blank), then all attributes associated with the specified variable are automatically deleted. 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`, `i`, `s`, `c`, `b`, `u`) or a short string standing for one of the twelve 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`.

`i, l` *Integer* or *Long*. Value(s) specified in *att_val* will be stored as netCDF intrinsic type `NC_INT`.

`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`.

`ub` *Unsigned Byte*. Value(s) specified in *att_val* will be stored as netCDF intrinsic type `NC_UBYTE`.

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

`u, ui, ul` *Unsigned Int*. Value(s) specified in *att_val* will be stored as netCDF intrinsic type `NC_UINT`.

`ll, int64` *Int64*. Value(s) specified in *att_val* will be stored as netCDF intrinsic type `NC_INT64`.

`ull, uint64`
Uint64. Value(s) specified in *att_val* will be stored as netCDF intrinsic type `NC_UINT64`.

`sng` *String*. Value(s) specified in *att_val* will be stored as netCDF intrinsic type `NC_STRING`.

The specification of *att_type* is optional (and is ignored) 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* (and is ignored) 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 single or double 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 what 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 4.7 [ncks netCDF Kitchen Sink], page 82, 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 -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 -a long_name,T,o,c,temperature in.nc
```

Delete all existing `units` attributes:

```
ncatted -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.

Delete all attributes associated with the `tpt` variable:

```
ncatted -a ,tpt,d,, in.nc
```

The value of *att_nm* was left blank in order to select all attributes associated with the variable. To delete all global attributes, simply replace `tpt` with `global` in the above.

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

```
ncatted -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.

4.3 ncbo netCDF Binary Operator

SYNTAX

```
ncbo [-4] [-A] [-C] [-c] [-D dbg]
[-d dim,[min][,[max][,[stride]]] [-F] [-h]
[-L dfl_lvl] [-l path] [-O] [-o file_3] [-p path] [-R] [-r]
[-t thr_nbr] [-v var[,...]] [-x] [-y op_typ]
file_1 file_2 [file_3]
```

DESCRIPTION

`ncbo` performs binary operations on variables in *file_1* and the corresponding variables (those with the same name) in *file_2* and stores the results in *file_3*. The binary operation operates on the entire files (modulo any excluded variables). See Section 3.19 [Missing Values], page 40, for treatment of missing values. One of the four standard arithmetic binary operations currently supported must be selected with the '`-y op_typ`' switch (or long options '`--op_typ`' or '`--operation`'). The valid binary operations for `ncbo`, their definitions, corresponding values of the *op_typ* key, and alternate invocations are:

Addition Definition: *file_3* = *file_1* + *file_2*
Alternate invocation: `ncadd`
op_typ key values: '`add`', '`+`', '`addition`'
Examples: '`ncbo --op_typ=add 1.nc 2.nc 3.nc`', '`ncadd 1.nc 2.nc 3.nc`'

Subtraction
Definition: *file_3* = *file_1* - *file_2*
Alternate invocations: `ncdiff`, `ncsub`, `ncsubtract`
op_typ key values: '`sbt`', '`-`', '`dff`', '`diff`', '`sub`', '`subtract`', '`subtraction`'
Examples: '`ncbo --op_typ=- 1.nc 2.nc 3.nc`', '`ncdiff 1.nc 2.nc 3.nc`'

Multiplication
Definition: *file_3* = *file_1* * *file_2*
Alternate invocations: `ncmult`, `ncmultiply`
op_typ key values: '`mlt`', '`*`', '`mult`', '`multiply`', '`multiplication`'
Examples: '`ncbo --op_typ=mlt 1.nc 2.nc 3.nc`', '`ncmult 1.nc 2.nc 3.nc`'

Division Definition: *file_3* = *file_1* / *file_2*
Alternate invocation: `ncdivide`
op_typ key values: '`dvd`', '`/`', '`divide`', '`division`'
Examples: '`ncbo --op_typ=/ 1.nc 2.nc 3.nc`', '`ncdivide 1.nc 2.nc 3.nc`'

Care should be taken when using the shortest form of key values, i.e., '`+`', '`-`', '`*`', and '`/`'. Some of these single characters may have special meanings to the shell [1]. Place these

[1] A naked (i.e., unprotected or unquoted) '`*`' is a wildcard character. A naked '`-`' may confuse the command line parser. A naked '`+`' and '`/`' are relatively harmless.

characters inside quotes to keep them from being interpreted (globbed) by the shell[2]. For example, the following commands are equivalent

```
ncbo --op_typ=* 1.nc 2.nc 3.nc # Dangerous (shell may try to glob)
ncbo --op_typ='*' 1.nc 2.nc 3.nc # Safe ('*' protected from shell)
ncbo --op_typ="*" 1.nc 2.nc 3.nc # Safe ('*' protected from shell)
ncbo --op_typ=mlt 1.nc 2.nc 3.nc
ncbo --op_typ=mult 1.nc 2.nc 3.nc
ncbo --op_typ=multiply 1.nc 2.nc 3.nc
ncbo --op_typ=multiplication 1.nc 2.nc 3.nc
ncmult 1.nc 2.nc 3.nc # First do 'ln -s ncbo ncmult'
ncmultiply 1.nc 2.nc 3.nc # First do 'ln -s ncbo ncmultiply'
```

No particular argument or invocation form is preferred. Users are encouraged to use the forms which are most intuitive to them.

Normally, `ncbo` will fail unless an operation type is specified with '`-y`' (equivalent to '`--op_typ`'). You may create exceptions to this rule to suit your particular tastes, in conformance with your site's policy on *symbolic links* to executables (files of a different name point to the actual executable). For many years, `ncdiff` was the main binary file operator. As a result, many users prefer to continue invoking `ncdiff` rather than memorizing a new command ('`ncbo -y sbt`') which behaves identically to the original `ncdiff` command. However, from a software maintenance standpoint, maintaining a distinct executable for each binary operation (e.g., `ncadd`) is untenable, and a single executable, `ncbo`, is desirable. To maintain backward compatibility, therefore, NCO automatically creates a symbolic link from `ncbo` to `ncdiff`. Thus `ncdiff` is called an *alternate invocation* of `ncbo`. `ncbo` supports many additional alternate invocations which must be manually activated. Should users or system adminitrators decide to activate them, the procedure is simple. For example, to use '`ncadd`' instead of '`ncbo --op_typ=add`', simply create a symbolic link from `ncbo` to `ncadd`[3]. The alternatate invocations supported for each operation type are listed above. Alternatively, users may always define '`ncadd`' as an *alias* to '`ncbo --op_typ=add`'[4].

It is important to maintain portability in NCO scripts. Therefore we recommend that site-specfic invocations (e.g., '`ncadd`') be used only in interactive sessions from the command-line. For scripts, we recommend using the full invocation (e.g., '`ncbo --op_typ=add`'). This ensures portability of scripts between users and sites.

`ncbo` operates (e.g., adds) variables in *file_2* with 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, but the reverse is not true. 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

[2] The widely used shell Bash correctly interprets all these special characters even when they are not quoted. That is, Bash does not prevent NCO from correctly interpreting the intended arithmetic operation when the following arguments are given (without quotes) to `ncbo`: '`--op_typ=+`', '`--op_typ=-`', '`--op_typ=*`', and '`--op_typ=/`'

[3] The command to do this is '`ln -s -f ncbo ncadd`'

[4] The command to do this is '`alias ncadd='ncbo --op_typ=add'`'

`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*. In the future, we will broadcast variables in *file_1*, if necessary to conform to their counterparts in *file_2*. Thus, presently, 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 initially the same dimensions as *file_1* (often *file_1* itself). In these cases, creating *file_2* with `ncra` rather than `ncwa` will cause the `ncbo` 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 [5]. In this case the input files to `ncbo`, *file_1* and *file_2*, will have unequally sized `time` dimensions which causes `ncbo` to fail. To prevent this from occuring, use `ncwa` to remove the `time` dimension from *file_2*. See the example below.

`ncbo` never operates on 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. `ncbo` applies special rules to some CF-defined (and/or NCAR CCSM or NCAR CCM fields) such as `ORO`. See Section 3.27 [CF Conventions], page 51 for a complete description. Finally, we note that `ncflint` (see Section 4.6 [ncflint netCDF File Interpolator], page 79) is designed for file interpolation. As such, it also performs file subtraction, addition, multiplication, albeit in a more convoluted way than `ncbo`.

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:

```
ncbo -op_typ=sub 86_0112.nc 85_0112.nc 86m85_0112.nc
ncdiff 86_0112.nc 85_0112.nc 86m85_0112.nc
```

The following examples demonstrate the broadcasting feature of `ncbo`. 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 `ncbo` to compute the anomalies for 1985:

[5] This is because `ncra` collapses the record dimension to a size of 1 (making it a *degenerate* dimension), but does not remove it, while, unless '`-b`' is given, `ncwa` removes all averaged dimensions. In other words, by default `ncra` changes variable size but not rank, while, `ncwa` changes both variable size and rank.

```
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 '85_x.nc', the zonal mean of '85.nc'. Then we use ncbo to subtract the zonal annual means from the monthly gridpoint data:

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

This examples works assuming '85_0112.nc' has dimensions time and lon, and that '85_x.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 five-year mean:

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

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

```
ncbo 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:

```
for idx in {01..12}; do # Bash Shell (version 3.0+, beware ordering!)
  ncks -F -d time,${idx},,12 t_anm_8501_8912.nc foo.${idx}
  ncra foo.${idx} t_anm_8589_${idx}.nc
done
for idx in 01 02 03 04 05 06 07 08 09 10 11 12; do # Bourne Shell
  ncks -F -d time,${idx},,12 t_anm_8501_8912.nc foo.${idx}
  ncra foo.${idx} t_anm_8589_${idx}.nc
done
foreach idx (01 02 03 04 05 06 07 08 09 10 11 12) # C Shell
  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
```

4.4 ncea netCDF Ensemble Averager

SYNTAX

```
ncea [-4] [-A] [-C] [-c] [-D dbg]
[-d dim,[min][,[max][,[stride]]] [-F] [-h] [-L dfl_lvl] [-l path]
[-n loop] [-O] [-o output-file] [-p path] [-R] [-r]
[-t thr_nbr] [-v var[,...]] [-x] [-y op_typ]
[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. `ncea` averages entire files, and weights each file evenly. This is distinct from `ncra`, which only averages over the record dimension (e.g., time), and weights each record in the record dimension evenly,

Variables in the *output-file* are the same size as the variable in each of the *input-files*, and all *input-files* must be the same size. The only exception is that `ncea` allows files to differ in the record dimension size if the requested record hyperslab (see Section 3.14 [Hyperslabs], page 33) resolves to the same size for all files. `ncea` recomputes the record dimension hyperslab limits for each input file so that coordinate limits may be used to select equal length timeseries from unequal length files. This simplifies analysis of unequal length timeseries from simulation ensembles (e.g., the IPCC AR4 archive).

`ncea` *always averages* coordinate variables regardless of the arithmetic operation type performed on the non-coordinate variables. (see Section 3.22 [Operation Types], page 43). All dimensions, including the record dimension, are treated identically and preserved in the *output-file*.

See Section 2.6 [Averaging vs. Concatenating], page 13, for a description of the distinctions between the various averagers and concatenators. As a multi-file operator, `ncea` will read the list of *input-files* from `stdin` if they are not specified as positional arguments on the command line (see Section 2.7 [Large Numbers of Files], page 14).

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 3.5 [Specifying Input Files], page 22, 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.

4.5 `ncecat` netCDF Ensemble Concatenator

SYNTAX

```
ncecat [-4] [-A] [-C] [-c] [-D dbg]
[-d dim,[min][,[max][,[stride]]] [-F] [-h] [-L dfl_lvl] [-l path]
[-n loop] [-O] [-o output-file] [-p path] [-R] [-r]
[-t thr_nbr] [-v var[,...]] [-x]
[input-files] [output-file]
```

DESCRIPTION

`ncecat` concatenates an arbitrary number of input files into a single output file. A new record dimension acts as the glue to bind the input file data together. Each variable in each input file becomes one record in the same variable in the output file. The *input-files* are stored consecutively as records in *output-file*. All *input-files* must contain all extracted variables (or else there would be "gaps" in the output file).

Each extracted variable must be constant in size and rank across all *input-files*. The only exception is that `ncecat` allows files to differ in the record dimension size if the requested record hyperslab (see Section 3.14 [Hyperslabs], page 33) resolves to the same size for all files. This allows easier gluing/averaging of unequal length timeseries from simulation ensembles (e.g., the IPCC AR4 archive).

Thus, the *output-file* size is the sum of the sizes of the extracted variables in the input files. See Section 2.6 [Averaging vs. Concatenating], page 13, for a description of the distinctions between the various averagers and concatenators. As a multi-file operator, `ncecat` will read the list of *input-files* from `stdin` if they are not specified as positional arguments on the command line (see Section 2.7 [Large Numbers of Files], page 14).

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 3.5 [Specifying Input Files], page 22, 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).

Consider a file with an existing record dimension named `time`. and suppose the user wishes to convert `time` from a record dimension to a non-record dimension. This may be useful, for example, when the user has another use for the record variable. The procedure is to use `ncecat` followed by `ncwa`

```
ncecat in.nc out.nc # Convert time to non-record dimension
ncwa -a record in.nc out.nc # Remove new degenerate record dimension
```

The second step removes the degenerate record dimension. See Section 4.8 [ncpdq netCDF Permute Dimensions Quickly], page 88 for other methods of changing variable dimensionality, including the record dimension.

4.6 `ncflint` netCDF File Interpolator

SYNTAX

```
ncflint [-4] [-A] [-C] [-c] [-D dbg]
[-d dim,[min][,[max][,[stride]]] [-F] [-h] [-i var,val3]
[-L dfl_lvl] [-l path] [-O] [-o file_3] [-p path] [-R] [-r]
[-t thr_nbr] [-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 is a weighted average, a normalized weighted 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 contributes to 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`' (or '`--weight`' or '`--wgt_var`') 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 $wgt2 = -3$. One can use this functionality to multiply all the values in a given file by a constant.

The second method of using `ncflint` is to specify the interpolation option with '`-i`' (or with the '`--ntp`' or '`--interpolate`' long options). 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. It makes sense to use this when the weights are known *a priori*.

Another class of problems has the *arrival value* (i.e., *val3*) of a particular variable *var* 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` then computes *wgt1* and *wgt2*, and uses 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 0.5`' or '`-w 0.5,0.5`'. Attempting to specify both '`-i`' and '`-w`' methods in the same command is an error.

`ncflint` does not interpolate variables of type `NC_CHAR` and `NC_BYTE`. This behavior is hardcoded.

Depending on your intuition, `ncflint` may treat missing values unexpectedly. Consider a point where the value in one input file, say *val1*, equals the missing value *mss_val_1* and, at the same point, the corresponding value in the other input file *val2* is not misssing (i.e., does not equal *mss_val_2*). There are three plausible answers, and this creates ambiguity.

Option one is to set *val3* = *mss_val_1*. The rationale is that `ncflint` is, at heart, an interpolator and interpolation involving a missing value is intrinsically undefined. `ncflint` currently implements this behavior since it is the most conservative and least likely to lead to misinterpretation.

Option two is to output the weighted valid data point, i.e., $val3 = wgt2 \times val2$. The rationale for this behavior is that interpolation is really a weighted average of known points, so `ncflint` should weight the valid point.

Option three is to return the *unweighted* valid point, i.e., *val3* = *val2*. This behavior would appeal to those who use `ncflint` to estimate data using the closest available data. When a point is not bracketed by valid data on both sides, it is better to return the known datum than no datum at all.

The current implementation uses the first approach, Option one. If you have strong opinions on this matter, let us know, since we are willing to implement the other approaches as options if there is enough interest.

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 0.667 85_01.nc 85_04.nc 85_02.nc
ncflint -w 0.667,0.333 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`'.

Add '`85.nc`' to '`86.nc`' to obtain '`85p86.nc`', then subtract '`86.nc`' from '`85.nc`' to obtain '`85m86.nc`'

```
ncflint -w 1,1 85.nc 86.nc 85p86.nc
```

```
ncflint -w 1,-1 85.nc 86.nc 85m86.nc
ncdiff 85.nc 86.nc 85m86.nc
```

Thus `ncflint` can be used to mimic some `ncbo` operations. However this is not a good idea in practice because `ncflint` does not broadcast (see Section 4.3 [ncbo netCDF Binary Operator], page 70) conforming variables during arithmetic. Thus the final two commands would produce identical results except that `ncflint` would fail if any variables needed to be broadcast.

Rescale the dimensional units of the surface pressure `prs_sfc` from Pascals to hectopascals (millibars)

```
ncflint -C -v prs_sfc -w 0.01,0.0 in.nc in.nc out.nc
ncatted -a units,prs_sfc,o,c,millibar out.nc
```

4.7 ncks netCDF Kitchen Sink

SYNTAX

```
ncks [-4] [-A] [-a] [-B] [-b binary-file] [-C] [-c]
[-D dbg] [-d dim,[min][,[max][,[stride]]]
[-F] [-H] [-h] [--hdr_pad nbr] [-L dfl_lvl] [-l path] [-M] [-m]
[-O] [-o output-file] [-P] [-p path] [-Q] [-q] [-R] [-r] [-s for-
mat] [-u]
[-v var[,...]] [-x] input-file [[output-file]]
```

DESCRIPTION

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

`ncks` will print netCDF data in ASCII format to `stdout`, like `ncdump`, 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`' (or long options '`--sng_fmt`' and '`--string`') lets the user format the data using C-style format strings.

Options '`-a`', '`-F`' , '`-H`', '`-M`', '`-m`', '`-P`', '`-Q`', '`-q`', '`-s`', and '`-u`' (and their long option counterparts) control the formatted appearance of the data.

`ncks` extracts (and optionally creates a new netCDF file comprised of) only selected variables from the input file (similar to the old `ncextr` specification). Only variables and coordinates may be specifically included or excluded—all global attributes and any attribute associated with an extracted variable are copied to the screen and/or output netCDF file. Options '`-c`', '`-C`', '`-v`', and '`-x`' (and their long option synonyms) control which variables are extracted.

`ncks` extracts hyperslabs from the specified variables (`ncks` implements the original `nccut` specification). Option '`-d`' controls the hyperslab specification. Input dimensions that are not associated with any output variable do not appear in the output netCDF. This feature removes superfluous dimensions from netCDF files.

`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. If naming conflicts exist between the two files, data in *output-file* is usually overwritten by the corresponding data from *input-file*. Thus, when appending, the user should backup *output-file* in case valuable data are inadvertantly 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 19.

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

'`-B 'file''`
: Activate native machine binary output writing to the default binary file, '`ncks.bnr`'. The `-B` switch is redundant when the `-b` '`file`' option is specified, and native binary output will be directed to the binary file '`file`'. Also '`--bnr`' and '`--binary`'. Writing packed variables in binary format is not supported.

'`-b 'file''`
: Activate native machine binary output writing to binary file '`file`'. Also '`--fl_bnr`' and '`--binary-file`'. Writing packed variables in binary format is not supported.

[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*.

'-d *dim*,[*min*][,[*max*][,[*stride*]]]'
Add *stride* argument to hyperslabber. For a complete description of the *stride* argument, See Section 3.15 [Stride], page 34.

'-H' Print data to screen. Also activated using '--print' or '--prn'. By default ncks prints all metadata and data to screen if no netCDF output file is specified. Use '-H' to print data to screen if a netCDF output is specified, or to restrict printing to data (no metadata) when no netCDF output is specified. Unless otherwise specified (with -s), each element of the data hyperslab prints 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 -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 -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 -C -v three_dmn_var 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. Also '--Mtd' and '--Metadata'. By default ncks prints global metadata to screen if no netCDF output file and no

variable extraction list is specified (with '`-v`'). Use '`-M`' to print global metadata to screen if a netCDF output is specified, or if a variable extraction list is specified (with '`-v`').

The various combinations of printing switches can be confusing. In an attempt to anticipate what most users want to do, `ncks` uses context-sensitive defaults for printing. Our goal is to minimize the use of switches required to accomplish the common operations. We assume that users creating a new file or overwriting (e.g., with '`-O`') an existing file usually wish to copy all global and variable-specific attributes to the new file. In contrast, we assume that users appending (e.g., with '`-A`' an explicit variable list from one file to another usually wish to copy only the variable-specific attributes to the output file. The switches '`-H`', '`-M`', and '`-m`' switches are implemented as toggles which reverse the default behavior. The most confusing aspect of this is that '`-M`' inhibits copying global metadata in overwrite mode and causes copying of global metadata in append mode.

```
ncks -O              in.nc out.nc # Copy   VAs and GAs
ncks -O       -v one in.nc out.nc # Copy   VAs and GAs
ncks -O -M    -v one in.nc out.nc # Copy   VAs not GAs
ncks -O    -m -v one in.nc out.nc # Copy   GAs not VAs
ncks -O -M -m -v one in.nc out.nc # Copy   only data (no atts)
ncks -A              in.nc out.nc # Append VAs and GAs
ncks -A       -v one in.nc out.nc # Append VAs not GAs
ncks -A -M    -v one in.nc out.nc # Append VAs and GAs
ncks -A    -m -v one in.nc out.nc # Append only data (no atts)
ncks -A -M -m -v one in.nc out.nc # Append GAs not VAs
```

where `VAs` and `GAs` denote variable and global attributes, respectively.

'`-m`' Print variable metadata to screen (similar to *`ncdump -h`*). This displays all metadata pertaining to each variable, one variable at a time. Also '`--mtd`' and '`--metadata`'. The `ncks` default behavior is to print variable metadata to screen if no netCDF output file is specified. Use '`-m`' to print variable metadata to screen if a netCDF output is specified.

'`-P`' Print data, metadata, and units to screen. The '`-P`' switch is a convenience abbreviation for '`-C -H -M -m -u`'. Also activated using '`--print`' or '`--prn`'. This set of switches is useful for exploring file contents.

'`-Q`' Toggle printing of dimension indices and coordinate values when printing arrays. Each variable's name appears flush left in the output. This helps locate specific variables in lists with many variables and different dimensions.

'`-q`' Turn off all printing to screen. This overrides the setting of all print-related switches, equivalent to *`-H -M -m`* when in single-file printing mode. When invoked with `-R` (see Section 3.8 [Retaining Retrieved Files], page 27), `ncks` automatically sets `-q`. This allows `ncks` to retrieve remote files without automatically trying to print them. Also '`--quiet`'.

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

'-u' Toggle the printing of a variable's units attribute, if any, with its values. Also '--units'.

EXAMPLES

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

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

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

```
ncks 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 3.25 [History Attribute], page 50) 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).

Copy all global attributes (and no variables) from 'in.nc' to 'out.nc':

```
ncks -A -x ~/nco/data/in.nc ~/out.nc
```

The '-x' switch tells NCO to use the complement of the extraction list (see Section 3.11 [Subsetting Variables], page 30). Since no extraction list is explicitly specified (with '-v'), the default is to extract all variables. The complement of all variables is no variables. Without any variables to extract, the append ('-A') command (see Section 2.4 [Appending Variables], page 12) has only to extract and copy (i.e., append) global attributes to the output file.

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 -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[1]=90 lev[2]=1000 lon[3]=270 three_dmn_var[23]=23
% ncks -s '%f\n' -C -v three_dmn_var in.nc
0.000000
1.000000
...
23.000000
% ncks -s '%+16.10f\n' -C -v three_dmn_var in.nc
   +0.0000000000
   +1.0000000000
...
  +23.0000000000
% ncks -s '%f, ' -C -v three_dmn_var in.nc
0.000000, 1.000000, ..., 23.000000,
```

The second and third options are useful when pasting data into text files like reports or papers. See Section 4.2 [ncatted netCDF Attribute Editor], page 65, 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 -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 $0.3 < band < 0.5$ 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 3.14 [Hyperslabs], page 33.

```
ncks -H -d time,5 -d lat,,0.0 -d lon,330.0, -d band,0.3,0.5
-d lev,1000.0 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,260.0,45.0 in.nc out.nc
```

For more details See Section 3.17 [Wrapped Coordinates], page 37.

4.8 ncpdq netCDF Permute Dimensions Quickly

SYNTAX

```
ncpdq [-4] [-A] [-a [-]dim[,...]] [-C] [-c] [-D dbg]
[-d dim,[min][,[max][,[stride]]] [-F] [-h] [-L dfl_lvl] [-l path]
[-M pck_map] [-O] [-o output-file] [-P pck_plc] [-p path]
[-R] [-r] [-t thr_nbr] [-U] [-v var[,...]] [-x]
input-file [output-file]
```

DESCRIPTION

`ncpdq` performs one of two distinct functions, packing or dimension permutation, but not both, when invoked. `ncpdq` is optimized to perform these actions in a parallel fashion with a minimum of time and memory. The *pdq* may stand for "Permute Dimensions Quickly", "Pack Data Quietly", "Pillory Dan Quayle", or other silly uses.

Packing and Unpacking Functions

The `ncpdq` packing (and unpacking) algorithms are described in Section 4.1.4 [Intrinsic functions], page 59, and are also implemented in `ncap2`. `ncpdq` extends the functionality of these algorithms by providing high level control of the *packing policy* so that users can pack (and unpack) entire files consistently with one command. The user specifies the desired packing policy with the '`-P`' switch (or its long option equivalents, '`--pck_plc`' and '`--pack_policy`') and its *pck_plc* argument. Four packing policies are currently implemented:

Packing (and Re-Packing) Variables [default]
Definition: Pack unpacked variables, re-pack packed variables
Alternate invocation: `ncpack`
pck_plc key values: '`all_new`', '`pck_all_new_att`'

Packing (and not Re-Packing) Variables
Definition: Pack unpacked variables, copy packed variables
Alternate invocation: none
pck_plc key values: '`all_xst`', '`pck_all_xst_att`'

Re-Packing Variables
Definition: Re-pack packed variables, copy unpacked variables
Alternate invocation: none
pck_plc key values: '`xst_new`', '`pck_xst_new_att`'

Unpacking
Definition: Unpack packed variables, copy unpacked variables
Alternate invocation: `ncunpack`
pck_plc key values: '`upk`', '`unpack`', '`pck_upk`'

Equivalent key values are fully interchangeable. Multiple equivalent options are provided to satisfy disparate needs and tastes of NCO users working with scripts and from the command line.

To reduce required memorization of these complex policy switches, `ncpdq` may also be invoked via a synonym or with switches that imply a particular policy. `ncpack` is a synonym for `ncpdq` and behaves the same in all respects. Both `ncpdq` and `ncpack` assume a default packing policy request of '`all_new`'. Hence `ncpack` may be invoked without any '`-P`' switch, unlike `ncpdq`. Similarly, `ncunpack` is a synonym for `ncpdq` except that `ncpack` implicitly assumes a request to unpack, i.e., '`-P pck_upk`'. Finally, the `ncpdq` '`-U`' switch (or its long option equivalents, '`--upk`' and '`--unpack`') requires no argument. It simply requests unpacking.

Given the menagerie of synonyms, equivalent options, and implied options, a short list of some equivalent commands is appropriate. The following commands are equivalent for packing: `ncpdq -P all_new`, `ncpdq --pck_plc=all_new`, and `ncpack`. The following commands are equivalent for unpacking: `ncpdq -P upk`, `ncpdq -U`, `ncpdq --pck_plc=unpack`, and `ncunpack`. Equivalent commands for other packing policies, e.g., '`all_xst`', follow by analogy. Note that `ncpdq` synonyms are subject to the same constraints and recommendations discussed in the secion on `ncbo` synonyms (see Section 4.3 [ncbo netCDF Binary Operator], page 70). That is, symbolic links must exist from the synonym to `ncpdq`, or else the user must define an `alias`.

The `ncpdq` packing algorithms must know to which type particular types of input variables are to be packed. The correspondence between the input variable type and the output, packed type, is called the *packing map*. The user specifies the desired packing map with the '`-M`' switch (or its long option equivalents, '`--pck_map`' and '`--map`') and its *pck_map* argument. Five packing maps are currently implemented:

Pack Floating Precisions to `NC_SHORT` *[default]*
Definition: Pack floating precision types to `NC_SHORT`
Map: Pack [`NC_DOUBLE`,`NC_FLOAT`] to `NC_SHORT`
Types copied instead of packed: [`NC_INT`,`NC_SHORT`,`NC_CHAR`,`NC_BYTE`]
pck_map key values: '`flt_sht`', '`pck_map_flt_sht`'

Pack Floating Precisions to `NC_BYTE`
Definition: Pack floating precision types to `NC_BYTE`
Map: Pack [`NC_DOUBLE`,`NC_FLOAT`] to `NC_BYTE`
Types copied instead of packed: [`NC_INT`,`NC_SHORT`,`NC_CHAR`,`NC_BYTE`]
pck_map key values: '`flt_byt`', '`pck_map_flt_byt`'

Pack Higher Precisions to `NC_SHORT`
Definition: Pack higher precision types to `NC_SHORT`
Map: Pack [`NC_DOUBLE`,`NC_FLOAT`,`NC_INT`] to `NC_SHORT`
Types copied instead of packed: [`NC_SHORT`,`NC_CHAR`,`NC_BYTE`]
pck_map key values: '`hgh_sht`', '`pck_map_hgh_sht`'

Pack Higher Precisions to `NC_BYTE`

Definition: Pack higher precision types to `NC_BYTE`
Map: Pack [`NC_DOUBLE`,`NC_FLOAT`,`NC_INT`,`NC_SHORT`] to `NC_BYTE`
Types copied instead of packed: [`NC_CHAR`,`NC_BYTE`]
pck_map key values: '`hgh_byt`', '`pck_map_hgh_byt`'

Pack to Next Lesser Precision

Definition: Pack each type to type of next lesser size
Map: Pack `NC_DOUBLE` to `NC_INT`. Pack [`NC_FLOAT`,`NC_INT`] to `NC_SHORT`. Pack `NC_SHORT` to `NC_BYTE`.
Types copied instead of packed: [`NC_CHAR`,`NC_BYTE`]
pck_map key values: '`nxt_lsr`', '`pck_map_nxt_lsr`'

The default '`all_new`' packing policy with the default '`flt_sht`' packing map reduces the typical `NC_FLOAT`-dominated file size by about 50%. '`flt_byt`' packing reduces an `NC_DOUBLE`-dominated file by about 87%.

The netCDF packing algorithm (see Section 4.1.4 [Intrinsic functions], page 59) is lossy—once packed, the exact original data cannot be recovered without a full backup. Hence users should be aware of some packing caveats: First, the interaction of packing and data equal to the *_FillValue* is complex. Test the `_FillValue` behavior by performing a pack/unpack cycle to ensure data that are missing *stay* missing and data that are not misssing do not join the Air National Guard and go missing. This may lead you to elect a new *_FillValue*. Second, `ncpdq` actually allows packing into `NC_CHAR` (with, e.g., '`flt_chr`'). However, the intrinsic conversion of `signed char` to higher precision types is tricky so for values equal to zero, i.e., `NUL`. Hence packing to `NC_CHAR` is not documented or advertised. Pack into `NC_BYTE` (with, e.g., '`flt_byt`') instead.

Dimension Permutation

`ncpdq` re-shapes variables in *input-file* by re-ordering and/or reversing dimensions specified in the dimension list. The dimension list is a whitespace-free, comma separated list of dimension names, optionally prefixed by negative signs, that follows the '`-a`' (or long options '`--arrange`', '`--permute`', '`--re-order`', or '`--rdr`') switch. To re-order variables by a subset of their dimensions, specify these dimensions in a comma-separated list following '`-a`', e.g., '`-a lon,lat`'. To reverse a dimension, prefix its name with a negative sign in the dimension list, e.g., '`-a -lat`'. Re-ordering and reversal may be performed simultaneously, e.g., '`-a lon,-lat,time,-lev`'.

Users may specify any permutation of dimensions, including permutations which change the record dimension identity. The record dimension is re-ordered like any other dimension. This unique `ncpdq` capability makes it possible to concatenate files along any dimension. See Section 2.6.1 [Concatenation], page 13 for a detailed example. The record dimension is always the most slowly varying dimension in a record variable (see Section 3.13 [C and Fortran Index Conventions], page 32). The specified re-ordering fails if it requires creating more than one record dimension amongst all the output variables[1].

[1] This limitation, imposed by the netCDF storage layer, may be relaxed in the future with netCDF4.

Two special cases of dimension re-ordering and reversal deserve special mention. First, it may be desirable to completely reverse the storage order of a variable. To do this, include all the variable's dimensions in the dimension re-order list in their original order, and prefix each dimension name with the negative sign. Second, it may useful to transpose a variable's storage order, e.g., from C to Fortran data storage order (see Section 3.13 [C and Fortran Index Conventions], page 32). To do this, include all the variable's dimensions in the dimension re-order list in reversed order. Explicit examples of these two techniques appear below.

NB: fxm ncpdq documentation will evolve through Fall 2004. I will upload updates to documentation linked to by the NCO homepage. ncpdq is a powerful operator, and I am unfamiliar with the terminology needed to describe what ncpdq does. Sequences, sets, sheesh! I just know that it does "The right thing" according to my gut feelings. Now do you feel more comfortable using it?

Let $\mathbf{D}(x)$ represent the dimensionality of the variable x. Dimensionality describes the order and sizes of dimensions. If x has rank N, then we may write $\mathbf{D}(x)$ as the N-element vector

$$\mathbf{D}(x) = [D_1, D_2, D_3, \ldots, D_{n-1}, D_n, D_{n+1}, \ldots, D_{N-2}, D_{N-1}, D_N]$$

where D_n is the size of the n'th dimension.

The dimension re-order list specified with '`-a`' is the R-element vector

$$\mathbf{R} = [R_1, R_2, R_3, \ldots, R_{r-1}, R_r, R_{r+1}, \ldots, R_{R-2}, R_{R-1}, R_R]$$

There need be no relation between N and R. Let the S-element vector $\mathbf{S}$ be the intersection (i.e., the ordered set of unique shared dimensions) of $\mathbf{D}$ and $\mathbf{R}$ Then

$$\begin{aligned}\mathbf{S} &= \mathbf{R} \cap \mathbf{D} \\ &= [S_1, S_2, S_3, \ldots, S_{s-1}, S_s, S_{s+1}, \ldots, S_{S-2}, S_{S-1}, S_S]\end{aligned}$$

$\mathbf{S}$ is empty if $\mathbf{R} \notin \mathbf{D}$.

Re-ordering (or re-shaping) a variable means mapping the input state with dimensionality $\mathbf{D}(x)$ to the output state with dimensionality $\mathbf{D}'(x')$. In practice, mapping occurs in three logically distinct steps. First, we tranlate the user input to a one-to-one mapping $\mathcal{M}$ between input and output dimensions, $\mathbf{D} \mapsto \mathbf{D}'$. This tentative map is final unless external constraints (typically netCDF restrictions) impose themselves. Second, we check and, if necessary, refine the tentative mapping so that the re-shaped variables will co-exist in the same file without violating netCDF-imposed storage restrictions. This refined map specifies the final (output) dimensionality. Third, we translate the output dimensionality into one-dimensional memory offsets for each datum according to the C language convention for multi-dimensional array storage. Dimension reversal changes the ordering of data, but not the dimensionality, and so is part of the third step.

Dimensions R disjoint from $\mathbf{D}$ play no role in re-ordering. The first step taken to re-order a variable is to determine $\mathbf{S}$. $\mathbf{R}$ is constant for all variables, whereas $\mathbf{D}$, and hence $\mathbf{S}$, is variable-specific. $\mathbf{S}$ is empty if $\mathbf{R} \notin \mathbf{D}$. This may be the case for some extracted variables. The user may explicitly specify the one-to-one mapping of input to output dimension order

by supplying (with '`-a`') a re-order list **R** such that $S = N$. In this case $D'_n = S_n$. The degenerate case occurs when $\mathbf{D} = \mathbf{S}$. This produces the identity mapping $D'_n = D_n$.

The mapping of input to output dimension order is more complex when $S \neq N$. In this case $D'_n = D_n$ for the $N - S$ dimensions $D'_n \notin \mathbf{S}$. For the S dimensions $D'_n \in \mathbf{S}$, $D'_n = S_s$.

EXAMPLES

Pack and unpack all variables in file '`in.nc`' and store the results in '`out.nc`':

```
ncpdq in.nc out.nc # Same as ncpack in.nc out.nc
ncpdq -P all_new -M flt_sht in.nc out.nc # Defaults
ncpdq -P all_xst in.nc out.nc
ncpdq -P upk in.nc out.nc # Same as ncunpack in.nc out.nc
ncpdq -U in.nc out.nc # Same as ncunpack in.nc out.nc
```

The first two commands pack any unpacked variable in the input file. They also unpack and then re-pack every packed variable. The third command only packs unpacked variables in the input file. If a variable is already packed, the third command copies it unchanged to the output file. The fourth and fifth commands unpack any packed variables. If a variable is not packed, the third command copies it unchanged.

The previous examples all utilized the default packing map. Suppose you wish to archive all data that are currently unpacked into a form which only preserves 256 distinct values. Then you could specify the packing map *pck_map* as '`hgh_byt`' and the packing policy *pck_plc* as '`all_xst`':

```
ncpdq -P all_xst -M hgh_byt in.nc out.nc
```

Many different packing maps may be used to construct a given file by performing the packing on subsets of variables (e.g., with '`-v`') and using the append feature with '`-A`' (see Section 2.4 [Appending Variables], page 12).

Re-order file '`in.nc`' so that the dimension `lon` always precedes the dimension `lat` and store the results in '`out.nc`':

```
ncpdq -a lon,lat in.nc out.nc
ncpdq -v three_dmn_var -a lon,lat in.nc out.nc
```

The first command re-orders every variable in the input file. The second command extracts and re-orders only the variable `three_dmn_var`.

Suppose the dimension `lat` represents latitude and monotonically increases increases from south to north. Reversing the `lat` dimension means re-ordering the data so that latitude values decrease monotonically from north to south. Accomplish this with

```
% ncpdq -a -lat in.nc out.nc
% ncks -C -v lat in.nc
lat[0]=-90
lat[1]=90
% ncks -C -v lat out.nc
lat[0]=90
lat[1]=-90
```

This operation reversed the latitude dimension of all variables. Whitespace immediately preceding the negative sign that specifies dimension reversal may be dangerous. Quotes and long options can help protect negative signs that should indicate dimension reversal from being interpreted by the shell as dashes that indicate new command line switches.

```
ncpdq -a -lat in.nc out.nc # Dangerous? Whitespace before "-lat"
ncpdq -a "-lat" in.nc out.nc # OK. Quotes protect "-" in "-lat"
ncpdq -a lon,-lat in.nc out.nc # OK. No whitespace before "-"
ncpdq --rdr=-lat in.nc out.nc # Preferred. Uses "=" not whitespace
```

To create the mathematical transpose of a variable, place all its dimensions in the dimension re-order list in reversed order. This example creates the transpose of `three_dmn_var`:

```
% ncpdq -a lon,lev,lat -v three_dmn_var in.nc out.nc
% ncks -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
% ncks -C -v three_dmn_var out.nc
lon[0]=0 lev[0]=100 lat[0]=-90 three_dmn_var[0]=0
lon[0]=0 lev[0]=100 lat[1]=90 three_dmn_var[1]=12
lon[0]=0 lev[1]=500 lat[0]=-90 three_dmn_var[2]=4
...
lon[3]=270 lev[1]=500 lat[1]=90 three_dmn_var[21]=19
lon[3]=270 lev[2]=1000 lat[0]=-90 three_dmn_var[22]=11
lon[3]=270 lev[2]=1000 lat[1]=90 three_dmn_var[23]=23
```

To completely reverse the storage order of a variable, include all its dimensions in the re-order list, each prefixed by a negative sign. This example reverses the storage order of `three_dmn_var`:

```
% ncpdq -a -lat,-lev,-lon -v three_dmn_var in.nc out.nc
% ncks -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
% ncks -C -v three_dmn_var out.nc
lat[0]=90 lev[0]=1000 lon[0]=270 three_dmn_var[0]=23
lat[0]=90 lev[0]=1000 lon[1]=180 three_dmn_var[1]=22
lat[0]=90 lev[0]=1000 lon[2]=90 three_dmn_var[2]=21
...
```

```
lat[1]=-90 lev[2]=100 lon[1]=180 three_dmn_var[21]=2
lat[1]=-90 lev[2]=100 lon[2]=90 three_dmn_var[22]=1
lat[1]=-90 lev[2]=100 lon[3]=0 three_dmn_var[23]=0
```

Consider a file with all dimensions, including `time`, fixed (non-record). Suppose the user wishes to convert `time` from a fixed dimension to a record dimension. This may be useful, for example, when the user wishes to append additional time slices to the data. The procedure is to use `ncecat` followed by `ncpdq` and then `ncwa`:

```
ncecat -O in.nc out.nc # Add degenerate record dimension named "record"
ncpdq -O -a time,record out.nc out.nc # Switch "record" and "time"
ncwa -O -a record out.nc out.nc # Remove (degenerate) "record"
```

The first step creates a degenerate (size equals one) record dimension. The second step swaps the ordering of the dimensions named `time` and `record`. Since `time` now occupies the position of the first (least rapidly varying) dimension, it becomes the record dimension. The dimension named `record` is no longer a record dimension. The third step averages over the degenerate `record` dimension. Averaging over a degenerate dimension does not alter the data. The ordering of other dimensions in the file (`lat`, `lon`, etc.) is immaterial to this procedure. See Section 4.5 [ncecat netCDF Ensemble Concatenator], page 77 for other methods of changing variable dimensionality, including the record dimension.

4.9 ncra netCDF Record Averager

SYNTAX

```
ncra [-4] [-A] [-C] [-c] [-D dbg]
[-d dim,[min][,[max][,[stride]]] [-F] [-h] [-L dfl_lvl] [-l path]
[-n loop] [-O] [-o output-file] [-p path] [-R] [-r]
[-t thr_nbr] [-v var[,...]] [-x] [-y op_typ]
[input-files] [output-file]
```

DESCRIPTION

`ncra` averages record variables across an arbitrary number of *input-files*. The record dimension is, by default, retained as a degenerate (size 1) dimension in the output variables. See Section 2.6 [Averaging vs. Concatenating], page 13, for a description of the distinctions between the various averagers and concatenators. As a multi-file operator, `ncra` will read the list of *input-files* from `stdin` if they are not specified as positional arguments on the command line (see Section 2.7 [Large Numbers of Files], page 14).

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 work correctly. `ncra` supports the *stride* argument to the '`-d`' hyperslab option (see Section 3.14 [Hyperslabs], page 33) 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. `ncra` *always averages* coordinate variables regardless of the arithmetic operation type performed on the non-coordinate variables. (see Section 3.22 [Operation Types], page 43).

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 3.5 [Specifying Input Files], page 22, 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 3.15 [Stride], page 34, 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
```

4.10 `ncrcat` netCDF Record Concatenator

SYNTAX

```
ncrcat [-4] [-A] [-C] [-c] [-D dbg]
[-d dim,[min][,[max][,[stride]]] [-F] [-h] [-L dfl_lvl] [-l path]
[-n loop] [-O] [-o output-file] [-p path] [-R] [-r]
[-t thr_nbr] [-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 2.6 [Averaging vs. Concatenating], page 13, for a description of the distinctions between the various averagers and concatenators. As a multi-file operator, `ncrcat` will read the list of *input-files* from `stdin` if they are not specified as positional arguments on the command line (see Section 2.7 [Large Numbers of Files], page 14).

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 3.28 [ARM Conventions], page 52 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 3.5 [Specifying Input Files], page 22, 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. When `ncra` and `ncrcat` encounter a file which does contain any records that meet the specified hyperslab criteria, they disregard the file and proceed to the next file without failing. 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 3.15 [Stride], page 34, 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
```

4.11 `ncrename` netCDF Renamer

SYNTAX

```
ncrename [-a old_name,new_name] [-a ...] [-D dbg]
[-d old_name,new_name] [-d ...] [-h] [--hdr_pad nbr] [-l path]
[-O] [-o output-file] [-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 old name is preceded by the character '`.`'. The validity of *old_name* is not checked prior to the renaming. Thus, if *old_name* is specified without the the '`.`' prefix and is not present in *input-file*, `ncrename` will abort. The *new_name* should never be prefixed by a '`.`' (the period will be included as part of the new name). The OPTIONS and EXAMPLES show how to select specific variables whose attributes are to be renamed.

`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 if the user makes a mistake. 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 with '`-a`' (or '`--attribute`') 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. As mentioned above, all occurrences of the attribute of a given name will be renamed unless the '`.`' form is used, with one exception. To change the attribute name for a particular variable, specify the *old_name* in the format *old_var_name*`@`*old_att_name*. The '`@`' symbol delimits the variable and attribute names. If the attribute is uniquely named (no other variables contain the attribute) then the *old_var_name*`@`*old_att_name* syntax is redundant. The *var_name*`@`*att_name* syntax is accepted, but not required, for the *new_name*.

'`-d old_name,new_name`'
: Dimension renaming. The old and new names of the dimension are specified with '`-d`' (or '`--dmn`', '`--dimension`') 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 with '`-v`' (or '`--variable`') 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
```

Rename the attribute `long_name` to `largo_nombre` in the variable `u`, and no other variables in netCDF '`in.nc`'.

```
ncrename -a u:long_name,largo_nombre in.nc
```

`ncrename` does not automatically attach dimensions to variables of the same name. If you want to rename a coordinate variable so that it remains a coordinate variable, you must separately rename both the dimension and the variable:

```
ncrename -d lon,longitude -v lon,longitude in.nc
```

Create netCDF '`out.nc`' identical to '`in.nc`' except the attribute `_FillValue` is changed to `missing_value`, the attribute `units` is changed to `CGS_units` (but only in those variables which possess it), the attribute `hieght` is changed to `height` in the variable `tpt`, and in the variable `prs_sfc`, if it exists.

```
ncrename -a _FillValue,missing_value -a .units,CGS_units \
  -a tpt@hieght,height -a prs_sfc@.hieght,height in.nc out.nc
```

The presence and absence of the '`.`' and '`@`' features cause this command to execute successfully only if a number of conditions are met. All variables *must* have a `_FillValue` attribute *and* `_FillValue` must also be a global attribute. The `units` attribute, on the other hand, will be renamed to `CGS_units` wherever it is found but need not be present in the file at all (either as a global or a variable attribute). The variable `tpt` must contain the `hieght` attribute. The variable `prs_sfc` need not exist, and need not contain the `hieght` attribute.

4.12 ncwa netCDF Weighted Averager

SYNTAX

```
ncwa [-4] [-A] [-a dim[,...]] [-B mask_cond] [-b] [-C] [-c] [-D dbg]
[-d dim,[min][,[max][,[stride]]] [-F] [-h] [-I] [-L dfl_lvl] [-l path]
[-M mask_val] [-m mask_var] [-N] [-n] [-O]
[-o output-file] [-p path] [-R] [-r] [-T mask_comp]
[-t thr_nbr] [-v var[,...]] [-W] [-w weight] [-x] [-y op_typ]
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 2.6 [Averaging vs. Concatenating], page 13, 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 to produce a scalar result for each.

Averaged dimensions are, by default, eliminated as dimensions. Their corresponding coordinates, if any, are output as scalars. The '`-b`' switch (and its long option equivalents '`--rdd`' and '`--degenerate-dimensions`') causes `ncwa` to retain averaged dimensions as degenerate (size 1) dimensions. This maintains the association between a dimension (or coordinate) and variables after averaging and simplifies, for instance, later concatenation along the degenerate dimension.

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 3.14 [Hyperslabs], page 33). `ncwa` also handles values matching the variable's `_FillValue` 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_var 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_var* 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_var* 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 45.0, the midpoint latitude of the hyperslab. Option '`-I`' overrides this default behavior and causes `ncwa` not to weight or mask coordinate variables[1]. In the above case, this causes the value of `latitude` in the *output-file* to be 45.0, an 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 arithmetic mean latitude of the averaged hyperslabs rather than the area-weighted mean latitude.[2].

As explained in See Section 3.22 [Operation Types], page 43, `ncwa` *always averages* coordinate variables regardless of the arithmetic operation type performed on the non-coordinate variables. This is independent of the setting of the '`-I`' option. The mathematical definition of operations involving rank reduction is given above (see Section 3.22 [Operation Types], page 43).

4.12.1 Mask 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_var* is specified (with '`-m`'). As noted above, *mask_var* is broadcast, if possible, to conform to the variable being averaged. In this case, the value of m_i depends on the *mask condition* also known as the *truth condition*. As expected, $m_i = 1$ when the mask condition is *true* and $m_i = 0$ otherwise.

The mask condition has the syntax *mask_var mask_comp mask_val*. The preferred method to specify the mask condition is in one string with the '`-B`' or '`--mask_condition`' switches. The older method is to use the three switches '`-m`', '`-T`', and '`-M`' to specify the *mask_var*, *mask_comp*, and *mask_val*, respectively.[3]. The *mask_condition* string is automatically parsed into its three constituents *mask_var*, *mask_comp*, and *mask_val*.

Here *mask_var* is the name of the masking variable (specified with '`-m`', '`--mask-variable`', '`--mask_variable`', '`--msk_nm`', or '`--msk_var`'). The truth *mask_comp* argument (specified with '`-T`', '`--mask_comparator`', '`--msk_cmp_typ`', or '`--op_rlt`' may be any one of the six arithmetic comparators: *eq*, *ne*, *gt*, *lt*, *ge*, *le*. These are the Fortran-style character abbreviations for the logical comparisons $=$, $\neq$, $>$, $<$, $\geq$, $\leq$. The mask comparator defaults to *eq* (equality). The *mask_val* argument to '`-M`' (or '`--mask-value`', or '`--msk_val`') is the right hand side of the *mask condition*. Thus for the *i*'th element of the hyperslab to be averaged, the mask condition is *mask$_i$ mask_comp mask_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 weighting variable *weight* (with '`-w`', '`--weight`', or '`--wgt_var`'). In this case, the value of w_i is determined by the *weight*

[1] The default behavior of ('`-I`') changed on 1998/12/01—before this date the default was not to weight or mask coordinate variables.

[2] 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.

[3] The three switches '`-m`', '`-T`', and '`-M`' are maintained for backward compatibility and may be deprecated in the future. It is safest to write scripts using '`--mask_condition`'.

variable in the *input-file*. 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 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 `_FillValue` 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).

4.12.2 Normalization and Integration

`ncwa` has one switch which controls the normalization of the averages appearing in the *output-file*. Short option '`-N`' (or long options '`--nmr`' or '`--numerator`') 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 arithmetic expression defining the operation (see Section 3.22 [Operation Types], page 43).

With this normalization option, `ncwa` can integrate variables. Averages are first computed as sums, and then normalized to obtain the average. The original sum (i.e., the numerator of the expression in Section 3.22 [Operation Types], page 43) is output if default normalization is turned off (with '`-N`'). This sum is the integral (not the average) over the specified (with '`-a`', or all, if none are specified) dimensions. The weighting variable, if specified (with '`-w`'), plays the role of the differential increment and thus permits more sophisticated integrals (i.e., weighted sums) to be output. For example, consider the variable `lev` where $lev = [100, 500, 1000]$ weighted by the weight `lev_wgt` where $lev\_wgt = [10, 2, 1]$. The vertical integral of `lev`, weighted by `lev_wgt`, is the dot product of *lev* and *lev_wgt*. That this is is 3000.0 can be seen by inspection and verified with the integration command

```
ncwa -N -a lev -v lev -w lev_wgt in.nc foo.nc;ncks foo.nc
```

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 and store results in '`out.nc`':

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

All variables in '`in.nc`' are reduced to scalars in '`out.nc`' since `ncwa` averages over all dimensions unless otherwise specified (with '`-a`').

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

```
ncwa -a lon in.nc out1.nc
ncwa -a lon -b in.nc out2.nc
```

The first command turns `lon` into a scalar and the second retains `lon` as a degenerate dimension in all variables.

```
% ncks -C -H -v lon out1.nc
lon = 135
% ncks -C -H -v lon out2.nc
lon[0] = 135
```

In either case the tally is simply the size of `lon`, i.e., for the '`85_0112.nc`' file described by the sample header above.

Compute the meridional (latitudinal) mean, with values weighted by the corresponding element of gw[4]:

```
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 mean 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-mean over the globe using only points for which $ORO < 0.5$[5]:

```
ncwa -B "ORO < 0.5"      -w gw -a lat,lon in.nc out.nc
ncwa -m ORO -M 0.5 -T lt -w gw -a lat,lon in.nc out.nc
```

[4] `gw` stands for *Gaussian weight* in many climate models.

[5] `ORO` stands for *Orography* in some climate models. $ORO < 0.5$ selects ocean-covered gridpoints.

It is considerably simpler to specify the complete *mask_cond* with the single string argument to '`-B`' than with the three separate switches '`-m`', '`-T`', and '`-M`'. If in doubt, enclose the *mask_cond* with double quotes since some of the comparators have special meanings to the shell.

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

Compute the global annual mean over the maritime tropical Pacific:

```
ncwa -B "ORO < 0.5"      -w gw -a lat,lon,time \
  -d lat,-20.0,20.0 -d lon,120.0,270.0 in.nc out.nc
ncwa -m ORO -M 0.5 -T lt -w gw -a lat,lon,time \
  -d lat,-20.0,20.0 -d lon,120.0,270.0 in.nc out.nc
```

Further examples will use the one-switch specification of *mask_cond*.

Determine the total area of the maritime tropical Pacific, assuming the variable *area* contains the area of each gridcell

```
ncwa -N -v area -B "ORO < 0.5" -a lat,lon \
  -d lat,-20.0,20.0 -d lon,120.0,270.0 in.nc out.nc
```

Weighting *area* (e.g., by *gw*) is not appropriate because *area* is *already* area-weighted by definition. Thus the '`-N`' switch, or, equivalently, the '`-y ttl`' switch, correctly integrate the cell areas into a total regional area.

Mask a file to contain *_FillValue* everywhere except where *thr_min* <= *msk_var* <= *thr_max*:

```
# Set masking variable and its scalar thresholds
export msk_var='three_dmn_var_dbl' # Masking variable
export thr_max='20' # Maximum allowed value
export thr_min='10' # Minimum allowed value
ncecat -O in.nc out.nc # Wrap out.nc in degenerate "record" dimension
ncwa -O -a record -B "${msk_var} <= ${thr_max}" out.nc out.nc
ncecat -O out.nc out.nc # Wrap out.nc in degenerate "record" dimension
ncwa -O -a record -B "${msk_var} >= ${thr_min}" out.nc out.nc
```

After the first use of `ncwa`, '`out.nc`' contains *_FillValue* where `${msk_var} >= ${thr_max}`. The process is then repeated on the remaining data to filter out points where `${msk_var} <= ${thr_min}`. The resulting '`out.nc`' contains valid data only where *thr_min* <= *msk_var* <= *thr_max*.

5 Contributing

We welcome contributions from anyone. The NCO project homepage at https://sf.net/projects/nco contains more information on how to contribute.

Financial contributions to NCO development may be made through PayPal. NCO has been shared for over 10 years yet only two users have contributed any money to the developers[1]. So you could be the third!

5.1 Contributors

The primary contributors to NCO development are:

Charlie Zender
: Concept, design and implementation of operators from 1995-2000. Since then autotools, bug-squashing, documentation, packing, NCO library redesign, `ncap2` features, `ncbo`, `ncpdq`, SMP threading and MPI parallelization, netCDF4 integration, project coordination, and releases.

Henry Butowsky
: Non-linear operations and `min()`, `max()`, `total()` support in `ncra` and `ncwa`. Type conversion for arithmetic. Migration to netCDF3 API. `ncap` parser, lexer, and I/O. Multislabbing algorithm. Variable wildcarding. Various hacks. `ncap2` language.

Rorik Peterson
: Autotool build support. Long command-line options. UDUnits support. Debianization. Numerous bug-fixes.

Harry Mangalam
: Benchmarking. OPeNDAP configuration.

Daniel Wang
: Script Workflow Analysis for MultiProcessing (SWAMP). RPM support.

Brian Mays
: Packaging for Debian GNU/Linux, `nroff` man pages.

George Shapovalov
: Packaging for Gentoo GNU/Linux.

Bill Kocik Memory management.

Len Makin
: NEC SX architecture support.

Jim Edwards
: AIX architecture support.

[1] Happy users have sent me a few gifts, though. This includes a box of imported chocolate. Mmm. Appreciation and gifts are definitely better than money. Naturally, I'm too lazy to split and send gifts to the other developers. However, unlike some NCO developers, I have a steady "real job". My intent is to split monetary donations among the active developers and to send them their shares via PayPal.

Juliana Rew
: Compatibility with large PIDs.

Scott Capps, Martin Dix, Mark Flanner, Keith Lindsay, Mike Page, Martin Schmidt, Michael Schulz, Remik Ziemlinski
: Excellent bug reports and feature requests.

5.2 Proposals for Institutional Funding

NSF has funded a project to improve Distributed Data Reduction & Analysis (DDRA) by evolving NCO into a suite of Scientific Data Operators called SDO. The two main components of this project are NCO parallelism (OpenMP, MPI) and Server-Side DDRA (SSDDRA) implemented through extensions to OPeNDAP and netCDF4. This project will dramatically reduce bandwidth usage for NCO DDRA.

With this first NCO proposal funded, the content of the next NCO proposal is clear. We are interested in obtaining NASA support for HDF-specific enhancements to NCO. We plan to submit a proposal to the next suitable NASA NRA or NSF opportunity.

We are considering a lot of interesting ideas for still more proposals. Please contact us if you wish to be involved with any future NCO-related proposals. Comments on the proposals and letters of support are also very welcome.

6 CCSM Example

This chapter presents an in depth example of using NCO to process and analyze the results of a CCSM climate simulation.

```
*******************************************************************************
Task 0: Finding input files
*******************************************************************************
The CCSM model outputs files to a local directory like:

/ptmp/zender/archive/T42x1_40

Each component model has its own subdirectory, e.g.,

/ptmp/zender/archive/T42x1_40/atm
/ptmp/zender/archive/T42x1_40/cpl
/ptmp/zender/archive/T42x1_40/ice
/ptmp/zender/archive/T42x1_40/lnd
/ptmp/zender/archive/T42x1_40/ocn

within which model output is tagged with the particular model name

/ptmp/zender/archive/T42x1_40/atm/T42x1_40.cam2.h0.0001-01.nc
/ptmp/zender/archive/T42x1_40/atm/T42x1_40.cam2.h0.0001-02.nc
/ptmp/zender/archive/T42x1_40/atm/T42x1_40.cam2.h0.0001-03.nc
...
/ptmp/zender/archive/T42x1_40/atm/T42x1_40.cam2.h0.0001-12.nc
/ptmp/zender/archive/T42x1_40/atm/T42x1_40.cam2.h0.0002-01.nc
/ptmp/zender/archive/T42x1_40/atm/T42x1_40.cam2.h0.0002-02.nc
...

or

/ptmp/zender/archive/T42x1_40/lnd/T42x1_40.clm2.h0.0001-01.nc
/ptmp/zender/archive/T42x1_40/lnd/T42x1_40.clm2.h0.0001-02.nc
/ptmp/zender/archive/T42x1_40/lnd/T42x1_40.clm2.h0.0001-03.nc
...

*******************************************************************************
Task 1: Regional processing
*******************************************************************************
The first task in data processing is often creating seasonal cycles.
Imagine a 100-year simulation with its 1200 monthly mean files.
Our goal is to create a single file containing 12 months of data.
Each month in the output file is the mean of 100 input files.

Normally, we store the "reduced" data in a smaller, local directory.
```

```
caseid='T42x1_40'
#drc_in="${DATA}/archive/${caseid}/atm"
drc_in="${DATA}/${caseid}"
drc_out="${DATA}/${caseid}"
mkdir -p ${drc_out}
cd ${drc_out}

Method 1: Assume all data in directory applies
for mth in {01..12}; do
  mm=`printf "%02d" $mth`
  ncra -O -D 1 -o ${drc_out}/${caseid}_clm${mm}.nc \
    ${drc_in}/${caseid}.cam2.h0.*-${mm}.nc
done # end loop over mth

Method 2: Use shell 'globbing' to construct input filenames
for mth in {01..12}; do
  mm=`printf "%02d" $mth`
  ncra -O -D 1 -o ${drc_out}/${caseid}_clm${mm}.nc \
    ${drc_in}/${caseid}.cam2.h0.00??-${mm}.nc \
    ${drc_in}/${caseid}.cam2.h0.0100-${mm}.nc
done # end loop over mth

Method 3: Construct input filename list explicitly
for mth in {01..12}; do
  mm=`printf "%02d" $mth`
  fl_lst_in=''
  for yr in {0001..0100}; do
    yyyy=`printf "%04d" $yr`
    fl_in=${caseid}.cam2.h0.${yyyy}-${mm}.nc
    fl_lst_in="${fl_lst_in} ${caseid}.cam2.h0.${yyyy}-${mm}.nc"
  done # end loop over yr
  ncra -O -D 1 -o ${drc_out}/${caseid}_clm${mm}.nc -p ${drc_in} \
    ${fl_lst_in}
done # end loop over mth

Make sure the output file averages correct input files!
ncks -M prints global metadata:

  ncks -M ${drc_out}/${caseid}_clm01.nc

The input files ncra used to create the climatological monthly mean
will appear in the global attribute named 'history'.

Use ncrcat to aggregate the climatological monthly means

  ncrcat -O -D 1 \
```

```
    ${drc_out}/${caseid}_clm??.nc ${drc_out}/${caseid}_clm_0112.nc

Finally, create climatological means for reference.
The climatological time-mean:

  ncra -O -D 1 \
    ${drc_out}/${caseid}_clm_0112.nc ${drc_out}/${caseid}_clm.nc

The climatological zonal-mean:

  ncwa -O -D 1 -a lon \
    ${drc_out}/${caseid}_clm.nc ${drc_out}/${caseid}_clm_x.nc

The climatological time- and spatial-mean:

  ncwa -O -D 1 -a lon,lat,time -w gw \
    ${drc_out}/${caseid}_clm.nc ${drc_out}/${caseid}_clm_xyt.nc

This file contains only scalars, e.g., "global mean temperature",
used for summarizing global results of a climate experiment.

Climatological monthly anomalies = Annual Cycle:
Subtract climatological mean from climatological monthly means.
Result is annual cycle, i.e., climate-mean has been removed.

  ncbo -O -D 1 -o ${drc_out}/${caseid}_clm_0112_anm.nc \
    ${drc_out}/${caseid}_clm_0112.nc ${drc_out}/${caseid}_clm_xyt.nc

*******************************************************************************
Task 2: Correcting monthly averages
*******************************************************************************
The previous step appoximates all months as being equal, so, e.g.,
February weighs slightly too much in the climatological mean.
This approximation can be removed by weighting months appropriately.
We must add the number of days per month to the monthly mean files.
First, create a shell variable dpm:

unset dpm # Days per month
declare -a dpm
dpm=(0 31 28.25 31 30 31 30 31 31 30 31 30 31) # Allows 1-based indexing

Method 1: Create dpm directly in climatological monthly means
for mth in {01..12}; do
  mm=‘printf "%02d" ${mth}‘
  ncap2 -O -s "dpm=0.0*date+${dpm[${mth}]}" \
    ${drc_out}/${caseid}_clm${mm}.nc ${drc_out}/${caseid}_clm${mm}.nc
done # end loop over mth
```

```
Method 2: Create dpm by aggregating small files
for mth in {01..12}; do
  mm=‘printf "%02d" ${mth}‘
  ncap2 -O -v -s "dpm=${dpm[${mth}]}" ~/nco/data/in.nc \
    ${drc_out}/foo_${mm}.nc
done # end loop over mth
ncecat -O -D 1 -p ${drc_out} -n 12,2,2 foo_${mm}.nc foo.nc
ncrename -O -D 1 -d record,time ${drc_out}/foo.nc
ncatted -O -h \
  -a long_name,dpm,o,c,"Days per month" \
  -a units,dpm,o,c,"days" \
  ${drc_out}/${caseid}_clm_0112.nc
ncks -A -v dpm ${drc_out}/foo.nc ${drc_out}/${caseid}_clm_0112.nc

Method 3: Create small netCDF file using ncgen
cat > foo.cdl << EOF
netcdf foo {
dimensions:
time=unlimited;
variables:
float dpm(time);
dpm:long_name="Days per month";
dpm:units="days";
data:
dpm=31,28.25,31,30,31,30,31,31,30,31,30,31;
}
EOF
ncgen -b -o foo.nc foo.cdl
ncks -A -v dpm ${drc_out}/foo.nc ${drc_out}/${caseid}_clm_0112.nc

Another way to get correct monthly weighting is to average daily
output files, if available.

*******************************************************************************
Task 3: Regional processing
*******************************************************************************
Let's say you are interested in examining the California region.
Hyperslab your dataset to isolate the appropriate latitude/longitudes.

  ncks -O -D 1 -d lat,30.0,37.0 -d lon,240.0,270.0 \
    ${drc_out}/${caseid}_clm_0112.nc ${drc_out}/${caseid}_clm_0112_Cal.nc

The dataset is now much smaller!
To examine particular metrics.

*******************************************************************************
```

```
Task 4: Accessing data stored remotely
***********************************************************************
OPeNDAP server examples:

UCI DAP servers:
ncks -M -p http://dust.ess.uci.edu/cgi-bin/dods/nph-dods/dodsdata in.nc
ncrcat -O -C -D 3 -p http://soot.ess.uci.edu/cgi-bin/dods/nph-dods/dodsdata \
  -l /tmp in.nc in.nc ~/foo.nc
ncrcat -O -C -D 3 -p http://dust.ess.uci.edu/cgi-bin/dods/nph-dods/dodsdata \
  -l /tmp in.nc in.nc ~/foo.nc

NOAA DAP servers:
ncwa -O -C -a lat,lon,time -d lon,-10.,10. -d lat,-10.,10. -l /tmp -p \
http://www.cdc.noaa.gov/cgi-bin/nph-nc/Datasets/ncep.reanalysis.dailyavgs/surface \
pres.sfc.1969.nc ~/foo.nc

PCMDI IPCC Data Portal:
ncks -M -p http://username:password@climate.llnl.gov/cgi-bin/dap-cgi.py/ipcc4/sresa1b/

Earth System Grid (ESG): http://www.earthsystemgrid.org

caseid='b30.025.ES01'
CCSM3.0 1% increasing CO2 run, T42_gx1v3, 200 years starting in year 400
Atmospheric post-processed data, monthly averages, e.g.,
/data/zender/tmp/b30.025.ES01.cam2.h0.TREFHT.0400-01_cat_0449-12.nc
/data/zender/tmp/b30.025.ES01.cam2.h0.TREFHT.0400-01_cat_0599-12.nc

ESG supports password-protected FTP access by registered users
NCO uses the .netrc file, if present, for password-protected FTP access
Syntax for accessing single file is, e.g.,
ncks -O -D 3 \
  -p ftp://climate.llnl.gov/sresa1b/atm/mo/tas/ncar_ccsm3_0/run1 \
  -l /tmp tas_A1.SRESA1B_1.CCSM.atmm.2000-01_cat_2099-12.nc ~/foo.nc

# Average surface air temperature tas for SRESA1B scenario
for var in 'tas'; do
for scn in 'sresa1b'; do
for mdl in 'cccma_cgcm3_1 cccma_cgcm3_1_t63 cnrm_cm3 csiro_mk3_0 \
gfdl_cm2_0 gfdl_cm2_1 giss_aom giss_model_e_h giss_model_e_r \
iap_fgoals1_0_g inmcm3_0 ipsl_cm4 miroc3_2_hires miroc3_2_medres \
miub_echo_g mpi_echam5 mri_cgcm2_3_2a ncar_ccsm3_0 ncar_pcm1 \
ukmo_hadcm3 ukmo_hadgem1'; do
for run in '1'; do
        ncks -R -O -D 3 -p ftp://climate.llnl.gov/${scn}/atm/mo/${var}/${mdl}/run${run
done # end loop over run
done # end loop over mdl
done # end loop over scn
```

```
done # end loop over var

cd sresa1b/atm/mo/tas/ukmo_hadcm3/run1/
ncks -H -m -v lat,lon,lat_bnds,lon_bnds -M tas_A1.nc | m
bds -x 096 -y 073 -m 33 -o ${DATA}/data/dst_3.75x2.5.nc # ukmo_hadcm3
ncview ${DATA}/data/dst_3.75x2.5.nc

# msk_rgn is California mask on ukmo_hadcm3 grid
# area is correct area weight on ukmo_hadcm3 grid
ncks -A -v area,msk_rgn ${DATA}/data/dst_3.75x2.5.nc \
${DATA}/sresa1b/atm/mo/tas/ukmo_hadcm3/run1/area_msk_ukmo_hadcm3.nc

Template for standardized data:
${scn}_${mdl}_${run}_${var}_${yyyymm}_${yyyymm}.nc

e.g., raw data
${DATA}/sresa1b/atm/mo/tas/ukmo_hadcm3/run1/tas_A1.nc
becomes standardized data

Level 0: raw from IPCC site--no changes except for name
         Make symbolic link name match raw data
Template: ${scn}_${mdl}_${run}_${var}_${yyyymm}_${yyyymm}.nc

ln -s -f tas_A1.nc sresa1b_ukmo_hadcm3_run1_tas_200101_209911.nc
area_msk_ukmo_hadcm3.nc

Level I: Add all variables (but not standardized in time)
         to file containing msk_rgn and area
Template: ${scn}_${mdl}_${run}_${yyyymm}_${yyyymm}.nc

/bin/cp area_msk_ukmo_hadcm3.nc sresa1b_ukmo_hadcm3_run1_200101_209911.nc
ncks -A -v tas sresa1b_ukmo_hadcm3_run1_tas_200101_209911.nc \
               sresa1b_ukmo_hadcm3_run1_200101_209911.nc
ncks -A -v pr  sresa1b_ukmo_hadcm3_run1_pr_200101_209911.nc \
               sresa1b_ukmo_hadcm3_run1_200101_209911.nc

If already have file then:
mv sresa1b_ukmo_hadcm3_run1_200101_209911.nc foo.nc
/bin/cp area_msk_ukmo_hadcm3.nc sresa1b_ukmo_hadcm3_run1_200101_209911.nc
ncks -A -v tas,pr foo.nc sresa1b_ukmo_hadcm3_run1_200101_209911.nc

Level II: Correct # years, months
Template: ${scn}_${mdl}_${run}_${var}_${yyyymm}_${yyyymm}.nc

ncks -d time,..... file1.nc file2.nc
ncrcat file2.nc file3.nc sresa1b_ukmo_hadcm3_run1_200001_209912.nc
```

```
Level III: Many derived products from level II, e.g.,

      A. Global mean timeseries
      ncwa -w area -a lat,lon \
           sresa1b_ukmo_hadcm3_run1_200001_209912.nc \
   sresa1b_ukmo_hadcm3_run1_200001_209912_xy.nc

      B. Califoria average timeseries
      ncwa -m msk_rgn -w area -a lat,lon \
           sresa1b_ukmo_hadcm3_run1_200001_209912.nc \
   sresa1b_ukmo_hadcm3_run1_200001_209912_xy_Cal.nc
```

General Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Table of Contents