SourceForge.net Logo | **SF Tcl** | **SF Tk** | **SF BLT** | **SF NAP**

Tcl-nap Project

This is the home page (http://tcl-nap.sourceforge.net/) for the **Tcl-nap** SourceForge Project. This project provides the **NAP** *(n-dimensional array processor)* extension to Tcl.

Tcl-nap project resources at SourceForge

- Tcl-nap Project Summary
- Files for Downloading
- Browse CVS Source Repository
- Using CVS to access CVS Source Repository
- Bug Database
- Patch Manager

E-mail

- Mailing list for users of NAP
- Mailing list for developers of NAP
- Mailing list for announcements of new versions of NAP
- Harvey Davies (main developer of NAP)

Documentation

- Main NAP Documentation
- Talk at 9th Annual Tcl/Tk Conference, 2002 (revised version)
- CAPS (CSIRO project which spawned NAP)
- Tcl/Tk

Contents

Introduction

Background

NAP Commands

NAP Expressions

NAP Library of tcl code

NAP Internal Details and Interfacing

Overview of NAP (N-Dimensional Array Processor)

NAP is a loadable extension of Tcl which provides a powerful and efficient facility for processing data in the form of n-dimensional arrays. It has been designed to provide a tcl-flavoured array-processing facility with much of the functionality of languages such as APL, Fortran-90, IDL, J, matlab and octave. Three other tcl extensions which provide array-processing facilities are TiM, BLT and Tk3D.

Existing Tcl facilities (e.g. Tcl variables and procedures) are used where appropriate. The new facilities have been designed to match similar existing ones. In particular, NAP expressions use conventions which are essentially a superset of those of the Tcl `expr` command. Support is provided for data based on *n-dimensional grids*, where the dimensions correspond to continuous spatial coordinates. There are interfaces to the HDF and netCDF file formats commonly used for such data, especially in Earth sciences such as Oceanography and Meteorology. There is a new photo image format for NAP data.

NAP was developed as part of the CSIRO CAPS project, but can be loaded and used without the (satellite oriented) CAPS extension. However the CAPS extension requires NAP since most CAPS data are stored as NAOs.

NAP includes an interface to the BLT vector facility which is used by the NAP graphing procedure `plot_nao` to provide fast transfer of data to the BLT `graph` and `barchart` commands. Thus NAP must be linked with the binary libraries for BLT, HDF and netCDF.

Data are stored in memory as *n-dimensional array objects (NAOs)*, which include information such as:

- data-type
- unique ID (handle generated by NAP) called the *OOC-name* (OOCs are discussed below)
- optional label
- optional unit of measure
- reference count (allowing automatic deletion of the NAO when it is no longer needed)
- optional missing-value (used to indicate undefined data, etc.)
- rank (number of dimensions)
- dimension sizes
- optional dimension names
- optional pointers to *coordinate-variable* NAOs associated with each dimension

There are eleven data-types, six for integers, two for floating-point, one for characters, one *pointer* type (allowing arrays of arrays) and a *ragged* type providing a form of compression.

NAOs are created by the Tcl commands `nap` and `nap_get`.

The `nap` command takes arguments specifying an expression in a manner similar to the `expr` command. However, unlike `expr`, `nap` provides:

- array facilities (constants, operators, functions, indexing)
- assignment (to a Tcl variable whose value is set to the OOC-name of the resultant NAO)
- substitution of Tcl names (obviating the need for "`$`" prefixes)

The `nap_get` command creates a NAO from data read from a binary, HDF or netCDF file.

Every NAO has an associated Tcl command called an *object-oriented command* (*OOC*). This is used to:

- display the data in the NAO
- display other information about the NAO such as its data-type and dimensions
- change data and other details
- write data from the NAO to a binary, HDF or netCDF file
- write data from the NAO to a BLT vector

As usual in Tcl, OOC and `nap_get` command options can be abbreviated provided there is no ambiguity.

NAP provides many operators and built-in functions. One can define new functions by defining Tcl procedures with the `proc` command. It is also possible to call functions written in C and Fortran.

NAP Visualisation using procedure `plot_nao`

A NAO can be visualised using procedure `plot_nao`, which can represent it by:

- xy graphs (produced using BLT `graph` command)
- barcharts and histograms (produced using BLT `barchart` command)
- color-coded z images and maps (produced using the BLT graph image marker facility to overlay a NAP photo-image)
- tiled images (multiple images on a page) (produced directly from NAP photo-images)

It is possible to animate color-coded images and xy graphs.

The Tcl code is in the files `plot_nao.tcl` and `plot_nao_procs.tcl`.

Usage

`plot_nao` *expression ?options?*

Options

`-barwidth` *float*: (bar chart only) width of bars in x-coordinate units. (Default: 1.0)

`-buttonCommand` *script*: executed when button pressed with z-plots (Default: "lappend Plot_nao::${window_id}::save [set Plot_nao::${window_id}::xyz]")

`-colors` *list*: Colors of graph elements (lines). (Default: black red green blue yellow orange purple grey aquamarine beige)

`-configure` *script*: Prefix graph-name to each line & execute after graph created

`-filename` *string*: Name of output PostScript file.

`-fill` 0 or 1: 1 = Scale PostScript to fill page. (Default: 0)

`-gap_height` *int*: height (pixels) of horizontal white gaps (Default: 20)

`-gap_width` *int*: width (pixels) of vertical white gaps (Default: 20)

`-geometry` *string*: If specified then use to create new toplevel window.

`-height` *int*: Desired height (screen units) Type xy/bar: Height of whole window (Default: automatic) Type z: Image height (can be "min max") (Default: NAO dim if within limits)

`-help:` Display this help page

`-key` *int*: width (pixels) of image-key. No key if 0 or blank. (Default: 30)

`-labels` *list*: Labels of graph elements (lines). (Default: Use element names i.e. y0, y1, y2, …)

`-menu` 0 or 1: 0 = Start with menu bar at top hidden. (Default: 1)

`-orientation` P, L or A: P = portrait, L = landscape, A = automatic (Default: A)

`-overlay` C, L, S, N or "E *expression*": Define overlay. C = coast, L = land, S = sea, N = none, E = expr (Default: N)

`-ovpal` *expression*: Overlay palette in same form as main palette (Default: black white red green blue)

`-palette` *expression*: Main palette defining color map for 2D image. This is matrix with 3 or 4 columns & up to 256 rows. If there are 4 columns then the first gives color indices in range 0 to 255. Values can be whole numbers in range 0 to 255 or fractional values from 0.0 to 1.0. "" = black-to-white. (Default: blue-to-red)

`-paperheight` *distance*: E.g. '11i' = 11 inch (Default: '297m' = 297 mm (A4))

`-paperwidth` *distance*: E.g. '8.5i' = 8.5 inch (Default: '210m' = 210 mm (A4))

`-parent` *string*: parent window (Default: "" i.e. create toplevel window)

`-print` 0 or 1: 1 = automatic printing (for batch processing) (Default: 0)

`-printer` *string*: name (Default: env(PRINTER) if defined, else any printer)

`-range` *expression*: defines scaling (Default: auto scaling)

`-rank` 1, 2 or 3: rank of sub-arrays to be displayed (Default: 3 <<< rank(data))

`-scaling` 0 or 1: 0 = Start with scaling widget hidden. (Default: 1)

`-symbols` *list*: One for each element. Draw at points. Can be plus, square, circle, cross, splus, scross, triangle (Default: "" i.e. none)

`-title` *string*: title (Default: NAO label (if any) else *expression*)

`-type` *string* plot-type ("bar", "xy" or "z") If rank is 1 then default type is "xy". If rank is 2 and n_rows <= 8 then default type is "xy". If rank is 2 and n_rows > 8 then default type is "z". If rank is 3 then type is "z" regardless of this option.

`-width` *int*: Desired width (screen units) Type xy/bar: Width of whole window (Default: automatic) Type z: Image width (can be "min max") (Default: NAO dim if within limits)

`-xflip` 0 or 1: Flip left-right? 0 = no, 1 = yes. (Default: 0)

`-xlabel` *string*: x-axis label (Default: name of last dimension)

`-yflip` 0, 1, ascending or geog: Flip upside down? 0 = no, 1 = yes, ascending = "if y ascending", geog = "if ascending & (y_dim_name = latitude or y_unit = degrees_north (or equivalent))" (Default: geog)

`-ylabel` *string*: y-axis label (Default: name of 2nd-last dimension)

The height and width used are limited by the screen dimensions.

Examples

```
plot_nao "reshape(0 .. 199, {200 200})" -geometry +0+0
nap "sales = {{2 5 1}{1 3 8}}"
nap "month = 3 .. 5"
$sales set coord "" month
plot_nao sales -colors "blue red" -symbols "plus cross" -type xy
$sales set label "Car sales"
set windowID [plot_nao sales -labels "Joe Mary" -type bar]
set pathName $windowID.graph
$pathName element configure Joe -foreground blue"
```

Installing NAP

Introduction

The **nap** extension requires the **BLT** extension. The graphics procedure `plot_nao` uses the **printer** extension. The following instructions explain how to install these three extensions with **Tcl/Tk**.

Note that at the time of writing (15 October 2002) NAP could not be used with Tcl 8.4 (released 19 September 2002) because no compatible version of BLT was available. The most recent version of BLT available was 2.4y which was available for Tcl 8.3. Alternatives to BLT (for graphing) are being investigated to overcome this dependency.

Installing Tcl/Tk

Download **Tcl/Tk** from ActiveState Tcl Downloads. Installation instructions are available from this site.

Installing BLT

Download **BLT** from SourceForge: The BLT Toolkit Project Summary. For windows, select and execute the `.exe` file matching your version of Tcl. For unix, select the most recent source file (with suffix `.tar.gz`), unpack it into a new directory using a command such as

```
zcat BLT*.tar.gz | tar -xvf -
```

then compile using the instructions in the file `INSTALL`.

Installing Printer

Download **print(er)** from Michael Schwartz's Tcl/Tk Extensions & Information Page. Select **Print**, then press the **i** information button. For windows, select the `.zip` file matching your version of Tcl and unzip it into directory `lib`. (There are instructions in file `printer/install.txt`.) For unix, select the link labelled **Unix**, giving file `printUnix.tar.gz` and unpack it into directory `lib` using a command such as

```
zcat printUnix.tar.gz | tar -xvf -
```

Installing nap

Download **nap** from Tcl-NAP Files. Binary installation files are available for:

Operating System	Processor
Irix 64	SGI MIPS
Linux	Intel 386
SunOS 5.8	Sun sparc
Windows	Intel 386

(There is also a source file for those who want to access the source without using CVS.) All these files have the extension `.tgz`, which means they were created using `tar`, then compressed using `gzip`. They can be directly unpacked using many recent versions of **tar** and Windows programs such as **WinZip**. Unix users can also use:

```
zcat nap3.0.4src.tgz | tar -xvf -
```

You should unpack into the Tcl root directory (parent of `bin` and `lib`).

It is convenient to include the following

```
package require nap
namespace import ::NAP::*
```

in your startup file, which is normally stored in your home directory with the following name:

Operating System	tclsh	wish	tkcon
Windows	`tclshrc.tcl`	`wishrc.tcl`	`tkcon.cfg`
Unix	`.tclshrc`	`.wishrc`	`.tkconrc`

NAP Sample Session

The following sample session illustrates some basic features of NAP. The input command lines begin with the standard Tcl prompt "`%`".

```
% nap "x = {2 2.5 5}"
::NAP::13-13
% nap "y = x * x"
::NAP::14-14
% $y
4 6.25 25
```

The first command assigns to `x` a vector containing the three elements 2, 2.5 and 5. The second command assigns to `y` a vector containing the three elements which are the squares of the corresponding elements of `x`. The command "`$y`" returns the value of `y`.

Nap stores each variable in memory using a data-structure called an *n-dimensional array object* (*NAO*). Each NAO has an associated Tcl command called its *object-oriented command* (*OOC*) which is used to

- obtain data and other information from the NAO
- write data from the NAO to files and internal structures
- modify the NAO.

An *OOC-name* (command-name of an OOC) is used
- to execute the OOC (like any other Tcl command)
- as a unique identifier for the NAO associated with the OOC.

An OOC-name has the form `::NAP::`*seq*`-`*slot*, where
- `::NAP::` is the Tcl namespace used by NAP
- *seq* is the *sequence number* assigned in order of creation
- *slot* is the index of an internal table used to provide fast access.

In the above example, both OOC-names (`::NAP::13-13` and `::NAP::14-14`) have slots equal to their sequence number, but this is not the case in general since the slots of deleted NAOs may be reused.

An assignment ("`=`") operator has on its left a standard Tcl variable name which is assigned the (string) value of the OOC-name. Continuing the above example, these string values can be displayed using the standard Tcl command `set`.

```
% set x
::NAP::13-13
% set y
```

```
  ::NAP::14-14
```

Thus the command "`$y`" is equivalent to the command "`::NAP::14-14`". Confirming this:

```
% ::NAP::14-14
    4 6.25 25
```

If an OOC has no arguments (as above) then it returns the value of the NAO (abbreviated if the NAO is large). Arguments can be specified as in:

```
  % $x all
::NAP::13-13  f64  MissingValue: NaN  References: 1  Unit: (NULL)
Dimension 0   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
2 2.5 5
```

This illustrates the "`all`" *method* (sub-command), which provides a more detailed description of the NAO than the default method. The following example uses the "`set value`" method to change the value of element 1 of `x` from 6.25 to 7.

```
% $x set value 7 1
% $x all
::NAP::13-13  f64  MissingValue: NaN  References: 1  Unit: (NULL)
Dimension 0   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
2 7 5
```

The similarity between the "`expr`" and "`nap`" commands for simple arithmetic is shown by:

```
% expr "2 * (1 - 0.25)"
1.5
% nap "2 * (1 - 0.25)"
::NAP::25-25
% ::NAP::25-25
```

```
1.5
% ::NAP::25-25
invalid command name "::NAP::25-25"
```

Note that the command "`::NAP::25-25`" worked the first time but failed when it was repeated. The NAOs reference count was zero, as it was not referenced by anything (e.g. a Tcl variable). So the NAO and its associated OOC were automatically deleted after the first execution of the OOC.

The need to type the additional command "`::NAP::25-25`" can be obviated using the Tcl bracket ("`[]`") notation. Tcl executes the bracketed command, substitutes its result and then executes the generated command. So the above can be replaced by:

```
% [nap "2 * (1 - 0.25)"]
1.5
```

The following example illustrates *array indexing* and the calculation of an *arithmetic-mean* (both directly and by defining a function). The first two commands:

- define a 32-bit floating-point vector containing the five values 56, 75, 47, 99 and 49
- assign it to the variable `score`
- display it

```
% nap "score = f32{56 75 47 99 49}"
::NAP::16-16
% $score all
::NAP::16-16  f32  MissingValue: NaN  References: 1  Unit: (NULL)
Dimension 0   Size: 5      Name: (NULL)    Coordinate-variable: (NULL)
Value:
56 75 47 99 49
```

The following four commands respectively illustrate:

1. indexing a vector by a scalar "`2`" to give a scalar
2. indexing a vector by a vector "`{2 0 4}`" to give a vector
3. the operator "`..`" which defines an *arithmetic progression*
4. the use of such an arithmetic progression as an index

```
% [nap "score(2)"] all
::NAP::20-20  f32  MissingValue: NaN  References: 0  Unit: (NULL)
Value:
47
% [nap "score({2 0 4})"] all
```

```
::NAP::25-25  f32  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
47 56 49
% [nap "0 .. 3"]
0 1 2 3
% [nap "score(0 .. 3)"]
56 75 47 99
```

The following three commands respectively illustrate:

1. function `sum`, which has the functionality of mathematical "Σ"
2. function `count`, which gives the *number of non-missing elements*
3. the use of these functions to calculate an *arithmetic-mean*

```
% [nap "sum(score)"]
326
% [nap "count(score)"]
5
% [nap "sum(score) / count(score)"]
65.2
```

The following two commands respectively illustrate:

1. the definition of a tcl procedure to calculate an arithmetic-mean using NAP
2. the calling of this procedure as a NAP function

```
% proc mean x {nap "sum(x)/count(x)"}
% [nap "mean(score)"]
65.2
```

Procedures defining NAP functions have arguments and results which are OOC-names. All the facilities of Tcl and NAP can be used. So recursion is allowed, as shown by the following *factorial* example:

```
% proc factorial n {
    if {[[nap "n > 1"]]} {
        nap "n * factorial(n-1)"
    } else {
        nap "1"
    }
}
% [nap "factorial(4)"]
```

```
24
```

Note the double brackets (inside braces) in the first line of the body of the above procedure. The inner brackets produce an OOC-name. The outer brackets execute this OOC to produce the string "`0`" or "`1`".

Typographic Conventions

Consider the following example:
`count(`*x*`[,`*r*`])`

The font used for "`count(,)`" indicates this is *literal text*. In other words this is exactly what appears on the screen (which could be either output or typed input). Note that "*x*" and "*r*" are in italics (with slanted font), which indicates these are *formal argument names* rather than literal text. You replace such names with whatever is desired.

Optional arguments are indicated using two alternative conventions. In the above example they are enclosed in brackets, indicating that "`,`*r*" is optional. This convention is common in computing documentation. Use has also been made of the the alternative (commonly used in Tcl documentation) of surrounding the optional component with question marks. For example:
`count(`*x* ?`,`*r*?`)`

Alternatives are indicated with a vertical bar "|", as in the following:
`nap_info bytes`|`sequence`
which indicates that the argument can be either "`bytes`" or "`sequence`".

Data Models

A *data model* is a mental model of the nature of some data. It answers such questions as the following:

What values can the data take?

Are they all numeric? Are they all integers? Is the set of possible values finite? What are the minimum and maximum possible values? Are ∞ and *NaN* possible values? Do some values have special meanings, such as indicating undefined or missing data?

What is the measurement level?

Data is often classified as follows according to *measurement level*:

Name of Level	Description	Valid Operations	Appropriate Measure of Central Tendency	Examples
nominal	values denote categories which have no order	= ≠	mode	color, postal code (e.g. zip) chemical species (e.g. CO_2)
ordinal	values are ordered differences meaningless	= ≠ < ≤ > ≥	median	Richter earthquake scale dates in form YYYYMMDD
interval	differences are valid quotients meaningless	= ≠ < ≤ > ≥ + –	arithmetic mean	temperature in °C
ratio	quotients are valid	= ≠ < ≤ > ≥ + – × ÷	geometric mean	temperature in °K

How accurate are the values?

A *measurement error* is the difference between the *true value* and the *measured value*. Measured values can differ from true values due to:

- finite precision of instrument
- systematic errors (e.g. inadequately calibrated instrument)
- random errors (due to finite size of sample)
- sampling errors (due to non-randomness of sample)
- blunders (e.g. a human misreading an instrument)

It is desirable to include error estimates with data.

Are the data located in some space?

A *time series* consists of values located along the time dimension. *Geographic* data is located along spatial dimensions such as latitude, longitude and altitude and may also have a time dimension. Note that longitude is *cyclic*.

The dimensions of the space can have a measurement level of *nominal*. For example, an accounting spreadsheet might have columns corresponding to *charge codes* and rows corresponding to *company divisions*.

Data located in a continuous space can be either *gridded* or *scattered*. Both types are discussed in NAP Grids.

Grids

Traditional array-processors, such as APL, are based on a data-model which has discrete dimensions whose corresponding subscripts take integer values. NAP handles such traditional arrays in the traditional manner. However NAP is based on a more general data-model which also allows non-integer subscript values. These values represent distances along dimensions which often correspond to the physical dimensions of spacio-temporal spaces, which are of course continuous.

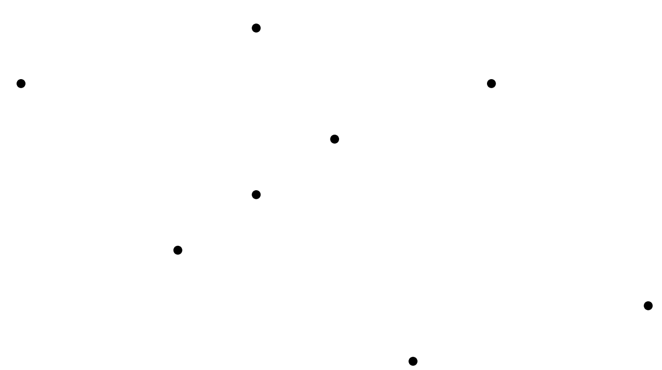
If desired, a dimension can have an associated variable called the *coordinate variable*. This is a vector which defines a piecewise-linear mapping from subscript to physical dimension.

Data in a continuous space with two or more dimensions can be either *gridded* or *scattered*. NAP's data-model (with continuous subscripts, coordinate variables, etc.) facilitates the processing of gridded data.

Let us consider the case of two dimensions i.e. matrices. Two-dimensional *gridded* data is aligned in rows and columns, whereas *scattered* data is not. The following examples are intended to illustrate the difference between *scattered* and *gridded* 2D data.

Example of Scattered 2D Data (Not a Grid)

Note that the data are not aligned in rows and columns.



Example of 2D Grid

The following example has the grid in black. The blue point is not on the grid and has non-integer subscript values (2.1, 1.6). The coordinate variables are latitude and longitude.

	column	0	1	1.6	2	3	4
	longitude	30°E	40°E	52°E	60°E	65°E	75°E

row	latitude							
0	30°N		•	•		•	•	•
1	25°N		•	•		•	•	•
2	10°N		•	•		•	•	•
2.1	8°N				•			
3	10°S		•	•		•	•	•

Missing Data

The NAP data model allows any element of an array to have a value which is a *missing-value*. Such elements are considered null or missing and are treated specially in operations such as arithmetic. Thus adding a missing value to anything produces a missing value.

The following example is similar to that above. However four grid points are missing. These are shown in red.

		column	0	1	1.6	2	3	4
		longitude	30°E	40°E	52°E	60°E	65°E	75°E
row	latitude							
0	30°N		•	•		•	•	•
1	25°N		•	•		•	•	•
2	10°N		•	•		•	•	•
2.1	8°N				•			
3	10°S		•	•		•	•	•

The missing points are treated as if they did not exist, as shown in the following:

		column	0	1	1.6	2	3	4
		longitude	30°E	40°E	52°E	60°E	65°E	75°E
row	latitude							
0	30°N		•	•		•		•
1	25°N		•	•		•	•	•
2	10°N			•		•	•	
2.1	8°N				•			

3	10°S		•	•		•	•

It would be possible to represent scattered data by a grid with many missing points. In the extreme, each scattered point would have its own row and its own column. There would be only one non-missing point in each row. There would be only one non-missing point in each column. Of course this would be very inefficient for a matrix of significant size.

Processing Scattered Data

NAP has no facilities for directly processing scattered data. However it is possible to define a grid from scattered data. This can be done by

- defining a least-squares best-fit n-dimensional surface and then using this to define the grid values
- using Delaunay triangulation (not yet implemented)

Terminology

Arrays

scalar
: array with 0 dimensions i.e. a simple number, character, etc.

vector
: array with 1 dimension.

matrix
: array with 2 dimensions.

dimension-size
: number of values along a dimension.

shape
: vector of dimension-sizes of array.

rank
: number of dimensions (a.k.a. dimensionality) i.e. shape of shape.

row
: final (least-significant) dimension

columns
: number of elements in each row

dimension-name
: name given to dimension e.g. "latitude".

coordinate-variable (CV)
: vector (usually sorted) associated with a dimension of the same size. CVs are often used to map an array's dimensions to physical dimensions such as length and time, thus locating the array elements in physical space and time.

Special Numeric Values

missing-value (MV)
: numeric value of data which is abnormal in some way such as:

- not applicable (e.g. land point for ocean data)
- not available (e.g. instrument failure or delay in obtaining data)
- result of some illegal operation such as dividing 0 by 0
- undefined for some other reason

infinity
: floating-point value representing a value which is too large (or small in the case of negative infinity) to represent. This can result from operations such as dividing by 0.

NaN (not-a-number)
: floating-point value resulting from an illegal operation such as dividing 0 by 0.

The nap command

The standard Tcl command "`expr`" is based on C conventions for operators and functions. NAP expressions use similar conventions and can include any of the following tokens separated by white-space characters:

- Operands
  - OOC-names
  - names of Tcl variables (may include namespaces)
  - constants
    - numeric scalar
    - numeric array
    - string
- Operators (including assignment operator "`=`")
- Parentheses ("`()`")
- Function names
  - built-in functions
  - names (may include namespaces) of Tcl procedures defining NAP functions
- Tcl substitution characters ("`[]$`")

If the `nap` command has multiple arguments then these are concatenated. Thus it is not always necessary to enclose the expression by quote (`"`) characters, but this practice is recommended because it

- prevents Tcl from removing braces (`{}`)
- allows standard Tcl (`$[]`) substitution within braces
- allows multi-line expressions.

Substitution

Like "`expr`", nap does the Tcl substitution defined by any brackets and dollars ("`[]$`") remaining after normal command parsing.

However, unlike "`expr`", nap also substitutes for Tcl variable names that are not preceded by a "`$`" (except where the name is the left operand of the assignment operator "`=`"). The value of the Tcl name is treated as a NAP expression, which is evaluated and the OOC-name of the result replaces the name. This substitution is repeated (up to eight times) until a single OOC-name is generated. The expressions in the following example include:

- Tcl variable `length` containing the string "`3.5`"
- Tcl variable `breadth` defined by NAP to contain "`::NAP::13-13`"
- NAP constants `2` and `10`

- Tcl variable `area` containing the string "`length * breadth`"

```
% set length 3.5
3.5
% nap "breadth = 2"
::NAP::13-13
% [nap "2 * (length + breadth)"]
11
% set area "length * breadth"
length * breadth
% [nap "10 * area"]
70
```

Each constant is replaced by the OOC-name of a NAO representing its value. After substitution, the expression consists of OOC-names, operators, function names and parentheses.

Namespaces

NAP allows names which include namespaces, as in:

```
% namespace eval ::mySpace {}; # create namespace "mySpace"
% nap "::mySpace::x = 8"
::NAP::13-13
% [nap "3 + ::mySpace::x"]
11
```

Functions

Function arguments can be enclosed by parentheses (as required by many other languages), but these parentheses are not required by the syntax. A name (which cannot be a Tcl variable name or it would have been substituted) followed by a OOC-name is treated as a function name. Thus the following two commands are equivalent:

```
% [nap "sin(3.14)"]
0.00159265
% [nap "sin 3.14"]
0.00159265
```

Indexing

Tcl array indices are enclosed by parentheses ("`()`"), while C uses brackets ("`[]`"). NAP requires neither, since indexing is simply implied by adjacent OOC-names. Thus the following two commands (which give elements 1, 0, 2 and 0 of the vector `{5 7 6}`) are equivalent:

```
% [nap "{5 7 6}({1 0 2 0})"]
7 5 6 5
% [nap "{5 7 6}{1 0 2 0}"]
7 5 6 5
```

Parenthesising Function Arguments and Indexes

As explained above, there is no syntactic need for parentheses around single function arguments and array indices. However, since most other computer languages do require such parentheses, it may aid human readability to include them.

Reading Files using `nap_get` Command

The `nap_get` command creates a NAO containing data read from a file. The first argument specifies the type of file, which can be `binary`, `hdf` or `netcdf`.

HDF and netCDF are similar array-oriented file formats which are popular in earth sciences such as meteorology and oceanography. (The new HDF5 format is not currently supported.) Such files contain data referenced by symbol tables containing the names, data-types and dimensions of variables. Each variable can also have attributes such as a label, a unit of measure and a missing-value. Note the similarity between these items and those in a NAO.

Reading Binary Data

Binary data is read using the command
`nap_get binary` *channel* [*datatype* [*shape*]]
where *datatype* defaults to `u8` and *shape* defaults to a vector corresponding to the file size.

The following example first writes six 32-bit floating-point values to a file using standard Tcl commands. It then reads them back into a NAO named "`in`" using "`nap_get binary`":

```
% set file [open float.dat w]
file6
% puts -nonewline $file [binary format f* {1.5 -3 0 2 4 5}]
% close $file
% set file [open float.dat]
file6
% nap "in = [nap_get binary $file f32]"
::NAP::22-22
% close $file
% $in all
::NAP::22-22  f32  MissingValue: NaN  References: 1  Unit: (NULL)
Dimension 0   Size: 6      Name: (NULL)    Coordinate-variable: (NULL)
Value:
1.5 -3 0 2 4 5
```

Note that no shape was specified, giving a 6-element vector. The following example reads the file again, this time specifying a shape of `{2 3}`. The NAO is displayed but not saved.

```
% set file [open float.dat]
file6
% [nap_get binary $file f32 "{2 3}"] all
::NAP::32-32  f32  MissingValue: NaN  References: 0  Unit: (NULL)
```

```
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
 1.5 -3.0  0.0
 2.0  4.0  5.0
% close $file
```

Reading netCDF Data

NetCDF data is read using the command
`nap_get netcdf` *filename name* [*index*]
name is the name of a variable or attribute and has the form

- *varname* for a variable
- *varname*`:`*attribute* for an attribute of a variable
- `:`*attribute* for a global attribute

If *index* is omitted then the entire variable is read in. Otherwise *index* selects using cross-product indexing.

The following example first creates a netCDF file using the netCDF utility `ncgen`. There is one variable called `vec`. It is a 3-element 32-bit integer vector with elements 6, -9 and 4. The data is read into a NAO called `v` using `nap_get netcdf`.

```
% exec ncgen -b << {
    netcdf int {
        dimensions:
            n = 3 ;
        variables:
            int vec(n) ;
        data:
            vec = 6, -9, 4 ;
    }
}
% nap "v = [nap_get netcdf int.nc vec]"
::NAP::52-52
% $v all
::NAP::52-52  i32  MissingValue: -2147483648  References: 1  Unit: (NULL)
Dimension 0   Size: 3      Name: n         Coordinate-variable: (NULL)
Value:
6 -9 4
```

The following example specifies the index `{0 2}` to select the first and third elements:

```
% [nap_get netcdf int.nc vec "{0 2}"] all
```

```
::NAP::58-58  i32  MissingValue: -2147483648  References: 0  Unit: (NULL)
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Value:
6 4
```

Reading HDF Data

HDF data is read using the command
`nap_get hdf` *filename name* [*index*]
name is the name of an SDS or attribute and has the form

- *sdsname* for a SDS
- *sdsname*`:`*attribute* for an attribute of a SDS
- `:`*attribute* for a global attribute

If *index* is omitted then the entire SDS is read in. Otherwise *index* selects using cross-product indexing.

The following example writes data to an HDF file using the OOC `hdf` method. Then `nap_get hdf` reads the data back into a temporary NAO which is deleted after being displayed:

```
% [nap "f64{{1 0 9}{3#-1}}"] hdf mat.hdf mat64
% [nap_get hdf mat.hdf mat64] all
::NAP::74-74  f64  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 2      Name: fakeDim0  Coordinate-variable: (NULL)
Dimension 1   Size: 3      Name: fakeDim1  Coordinate-variable: (NULL)
Value:
 1  0  9
-1 -1 -1
```

Listing Names of Variables/SDSs and Attributes in HDF and netCDF Files

One can list the names of variables/SDSs and attributes matching a regular expression *RE* using the command
`nap_get hdf -list` *filename* [*RE*]
or
`nap_get netcdf -list` *filename* [*RE*]
All variables/SDSs and attributes are listed if *RE* is omitted. For example, using the HDF file created above:

```
% nap_get hdf -list mat.hdf
mat64
mat64:_FillValue
```

Some useful regular expressions are

Regular Expression	Select all:
`^[^:]*$`	variables
`:`	attributes
`^:`	global attributes
`.:`	non-global attributes

Thus we can restrict the above list to SDSs only using:

```
% nap_get hdf -list mat.hdf {^[^:]*$}
mat64
```

Reading Metadata from HDF and netCDF Files

The command
`nap_get hdf -datatype` *filename sdsname*
or
`nap_get netcdf -datatype` *filename varname*
returns the data-type of a specified variable/SDS in the specified file.

The command
`nap_get hdf -rank` *filename sdsname*
or
`nap_get netcdf -rank` *filename varname*
returns the rank (number of dimensions) of a specified variable/SDS in the specified file.

The command
`nap_get hdf -shape` *filename sdsname*
or
`nap_get netcdf -shape` *filename varname*
returns the shape (dimension sizes) of a specified variable/SDS in the specified file.

The command
`nap_get hdf -dimension` *filename sdsname*
or
`nap_get netcdf -dimension` *filename varname*
returns the dimension names of a specified variable/SDS in the specified file.

The command
`nap_get hdf -coordinate` *filename sdsname*
or
`nap_get netcdf -coordinate` *filename varname dim_name|dim_number*

returns the name of the coordinate variable corresponding to a specified dimension of a specified variable/SDS in the specified file.

The `nap_info` Command

The `nap_info` command provides information about the NAP system. There are two variants which are shown in the following example:

```
% nap_info bytes
1300 1640
% nap_info sequence
15
```

The command "`nap_info bytes`" returns two numbers:

- current number of bytes of memory being used by NAOs (1300 in above example)
- maximum number of bytes used at any time so far(1640 in above example)

The command "`nap_info sequence`" returns the sequence-number of the most recently created NAO.

NAP Object-Oriented Commands (OOCs)

Methods which return Data Values (with or without metadata)

Method `all`

ooc_name `all` `-format` *format* `-columns` *int* `-lines` *int* `-missing` *text* `-keep`

This provides both data and metadata from a NAO. However it does not provide *all* information despite the name!

The following switches are allowed:
`-format` *format*: C format (default: "" meaning automatic)
`-columns` *int*: maximum # columns (default: 6) (`-1`: no limit)
`-lines` *int*: maximum # lines (default: 20) (`-1`: no limit)
`-list`: print in tcl list form (using braces) e.g. "`{1 9 2}`"
`-missing` *text*: text printed for missing value (default: "_")
`-keep`: Do not delete NAO with reference count of 0

The `all` method provides the same information as the two commands:
ooc_name `header`
ooc_name `value` `-format` *format* `-columns` *int* `-lines` *int* `-missing` *text*

Example:

```
% [nap "{3#2 2#_ -9}"] all -miss n/a
::NAP::39-39  i32  MissingValue: -2147483648  References: 0  Unit: (NULL)
Dimension 0   Size: 6      Name: (NULL)    Coordinate-variable: (NULL)
Value:
2 2 2 n/a n/a -9
```

Method `value`

ooc_name `value` `-format` *format* `-columns` *int* `-lines` *int* `-missing` *text* `-keep`

This returns data values. The default value is `-1` for both the switches `-columns` and `-lines`, giving the entire array.

The following switches are allowed:
`-format` *format*: C format (default: "" meaning automatic)
`-columns` *int*: maximum # columns (default: `-1` i.e. no limit)
`-lines` *int*: maximum # lines (default: `-1` i.e. no limit)
`-list`: print in tcl list form (using braces) e.g. "`{1 9 2}`"
`-missing` *text*: text printed for missing value (default: "_")

`-keep`: Do not delete NAO with reference count of 0

The following example begins with the definition of vectors "`x`" and "`y`" for use in this and subsequent examples:

```
% nap "x = 0 .. 2 ... 0.5"
::NAP::58-58
% nap "y = x ** 2"
::NAP::61-61
% $y val -format %0.3f
0.000 0.250 1.000 2.250 4.000
```

Default method

ooc_name `-format` *format* `-columns` *int* `-lines` *int* `-missing` *text* `-keep`

This returns data values in a similar fashion to the `value` method, except that default line and column limits restrict the size.

The following switches are allowed:
`-format` *format*: C format (default: `""` meaning automatic)
`-columns` *int*: maximum # columns (default: `6`) (`-1`: no limit)
`-lines` *int*: maximum # lines (default: `20`) (`-1`: no limit)
`-list`: print in tcl list form (using braces) e.g. "`{1 9 2}`"
`-missing` *text*: text printed for missing value (default: `"_"`)
`-keep`: Do not delete NAO with reference count of 0

The following example shows why and how to use switch `-columns` (abbreviated to `-c`):

```
% nap "m = reshape(0 .. 99, {10 12})"
::NAP::50-50
% $m
 0  1  2  3  4  5 ..
12 13 14 15 16 17 ..
24 25 26 27 28 29 ..
36 37 38 39 40 41 ..
48 49 50 51 52 53 ..
60 61 62 63 64 65 ..
72 73 74 75 76 77 ..
84 85 86 87 88 89 ..
96 97 98 99  0  1 ..
 8  9 10 11 12 13 ..
% $m -c -1
 0  1  2  3  4  5  6  7  8  9 10 11
12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31 32 33 34 35
```

```
36 37 38 39 40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69 70 71
72 73 74 75 76 77 78 79 80 81 82 83
84 85 86 87 88 89 90 91 92 93 94 95
96 97 98 99  0  1  2  3  4  5  6  7
 8  9 10 11 12 13 14 15 16 17 18 19
```

The following example shows how to use switch `-format` (abbreviated to `-f`) to include a dollar prefix and display two decimal places:

```
% [nap "{15 3.2 999}"] -f {$%.2f}
$15.00 $3.20 $999.00
```

The following example shows how to use switch `-list`:

```
% [nap "reshape(1.5 .. -1.5, {2 5})"] -list
{
{  1.5  0.5 -0.5 -1.5  1.5 }
{  0.5 -0.5 -1.5  1.5  0.5 }
}
```

Format Conversion Strings

The `-format` option specifies a *format conversion string* similar to that used in the standard Tcl `format` command (which is based on the ANSI C `sprintf()` function). Such strings have the form
%[*flags*][*width*][`.`*precision*]*char*
where:

- *flags* is a string containing any of the following characters in any order:
  `-`: left justify
  `+`: include sign
  space: include space prefix if no sign
  `0`: leading zeros
  #: alternate form
- *width* is the minimum field width
- *precision* is the
  - minimum number of digits for `d`, `i`, `o`, `x`, `X` or `u` conversions.
  - number of digits after "." for `e`, `E` or `f` conversions.
  - number of significant digits for `g` or `G` conversions.
- *char* specifies conversion as in the following table:

char	**Convert to**
`d, i`	signed decimal integer
`o`	unsigned octal integer

`x, X`	unsigned hexadecimal integer
`u`	unsigned decimal integer
`c`	character
`f`	decimal number in form [`-`]*mmm*`.`*ddd*, where number of *d*s is specified by the *precision*. Default *precision* is 6; *precision* of 0 suppresses the "`.`".
`e, E`	decimal number in form [`-`]*m*`.`*dddddd*`e`±*xx* or [`-`]*m*`.`*dddddd*`E`±*xx*, where number of *d*s is specified by the *precision*. Default *precision* is 6; *precision* of 0 suppresses the "`.`".
`g, G`	`%e` or `%E` are used if *exponent* < –4 or *exponent* ≥ *precision*; otherwise `%f` is used. Trailing zeros and decimal points are suppressed.

The following example displays the same data using each of these codes. Note that (unlike C and the standard Tcl `format` command) any data-type can be displayed with any code.

```
% foreach code {d i o x X u c f E e g G} {
    puts "$code: [[nap "88 .. 92"] -f "%$code"]"
}
d: 88 89 90 91 92
i: 88 89 90 91 92
o: 130 131 132 133 134
x: 58 59 5a 5b 5c
X: 58 59 5A 5B 5C
u: 88 89 90 91 92
c: XYZ[\
f: 88.000000 89.000000 90.000000 91.000000 92.000000
E: 8.800000E+01 8.900000E+01 9.000000E+01 9.100000E+01 9.200000E+01
e: 8.800000e+01 8.900000e+01 9.000000e+01 9.100000e+01 9.200000e+01
g: 88 89 90 91 92
G: 88 89 90 91 92
```

Methods which return Metadata

Method `coordinate`

ooc_name `coordinate` ?*dim_name*|*dim_number*? ?*dim_name*|*dim_number*? …

This returns the OOC-names of the coordinate variables of selected dimensions. If no dimensions are specified then the effect is the same as:
ooc_name `coo` 0 1 2 … rank-1"

Example:

```
% $y set coo x
% $y coo
::NAP::58-58
% [$y coo]
0 0.5 1 1.5 2
```

Method count

ooc_name `count` `-keep`

This returns the reference count.

Example (using `x` defined in previous example):

```
% $x count
2
```

Note that the reference count is 2 because this NAO is referenced by both
- Tcl variable `x`
- coordinate variable pointer of NAO `::NAP::61-61`

Method `datatype`

ooc_name `datatype`

This returns the data-type.

Example:

```
% [nap "'hello'"] dat
c8
```

Method `dimension`

ooc_name `dimension` ?*dim_number*? ?*dim_number*? …

This returns the dimension names.

ooc_name `di`
is equivalent to:
ooc_name `di` 0 1 2 … rank-1"

Example (again continuing above example):

```
% $y dim
x
```

Method `header`

ooc_name `header -keep`

This returns similar information to the following (but using a different format):
ooc_name `ooc`
ooc_name `datatype`
ooc_name `missing`
ooc_name `count`
ooc_name `unit`
ooc_name `shape`
ooc_name `dimension`
ooc_name `coordinate`

Example (continuing above example):

```
% $y header
::NAP::61-61  f64  MissingValue: NaN  References: 1  Unit: (NULL)
Dimension 0   Size: 5      Name: x         Coordinate-variable: ::NAP::58-58
%
```

Method `label`

ooc_name `label`

This returns the label (title, etc.) of the NAO.

Example:

```
% $y set label "areas of squares"
% $y label
areas of squares
```

Method `link`

ooc_name `link`

This returns the OOC-name of the link NAO.

Example:

```
% $y set link [nap 42]
% [$y link]
42
```

Method `missing`

ooc_name `missing`

This returns the missing value. This is the value used to indicate null or undefined data.

Example:

```
% $y miss
NaN
```

Method ooc

ooc_name `ooc -keep`

This returns the OOC-name of the NAO.

Example:

```
$y ooc
::NAP::61-61
```

Method `rank`

ooc_name `rank`

This returns the rank (number of dimensions).

Example:

```
% $y rank
1
```

Method sequence

ooc_name `sequence -keep`

This returns the sequence number of the NAO. E.g. 42 for `nao.42-9`

`-keep`: Do not delete NAO with reference count of 0

Example:

```
% $y seq
61
```

Method shape

ooc_name `shape`

This returns the shape, which is a vector of dimension sizes.

Example:

```
% $y shape
5
```

Method `slot`

ooc_name `slot -keep`

This returns the slot number of the NAO. E.g. 9 for `nao.42-9`

Example:

```
% $y sl
61
```

Method `step`

ooc_name `step`

This returns a code which indicates whether step sizes of a vector are equal, and if not, their sign. NAP uses this information for efficiency. It indicates whether a vector (not relevant for other ranks) is monotonically ascending/descending, and if so whether it is an arithmetic progression (AP). The result code is one of following strings:

- "`+-`": at least one positive step and one negative step
- "`>= 0`": all steps >= 0
- "`<= 0`": all steps <= 0
- "AP": equal steps (except final one which may be shorter)

Example:

```
% [nap "{3 5 7 7.1}"] step
AP
```

Method `unit`

ooc_name `unit`

This returns the unit of measure. This may be used in the future to support arithmetic with automatic unit conversion, but at the moment it is just descriptive information.

Example:

```
% $y set unit seconds
% $y unit
seconds
```

Methods which Modify NAO

Method `draw`

ooc_name `draw` *xy* ?*value*?

This draws a polyline in the NAO *ooc_name*, which must be type `f32` in the current version. It sets data elements on the polyline defined by NAO *xy* to the value of scalar NAO *value* (default: missing value). NAO *xy* can be:

- matrix with 2 rows, row 0 is x values, row 1 is y values
- vector of y values with coordinate variable (CV) of x values
- vector of y values without CV (x defaults to 0 1 2 3 …)

The polyline is not closed, so to draw a polygon, the first point should be duplicated at the end.

Example:

```
% [nap "z = reshape(0f32, 2 # 5)"] draw y 1
% $z
1 0 0 0 0
0 1 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 1 0 0
```

Method `fill`

ooc_name `fill` *xy* ?*value*?

This fills a polygon in the NAO *ooc_name*, which must be type `f32` in the current version. It sets data elements within the polygon defined by NAO *xy* to the value of the scalar NAO *value* (default: missing value).
NAO *xy* can be:

- matrix with 2 rows, row 0 is x values, row 1 is y values
- vector of y values with coordinate variable (CV) of x values
- vector of y values without CV (x defaults to 0 1 2 3 …)

The polygon is closed (unlike `draw` method).

Example:

```
% [nap "z = reshape(0f32, 2 # 8)"] fill "{{2 2 4 4}{2 4 4 2}}" 1
% $z value
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 1 1 1 0 0 0
0 0 1 1 1 0 0 0
0 0 1 1 1 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

Method `set`

ooc_name `set` *attribute arg arg …*

This modifies the NAO attribute specified by *attribute*. The sub-methods corresponding to these attributes are discussed below in separate sections. Note that the names of these attributes is the same as that of the method which returns their value.

`set coordinate`

ooc_name `set coordinate` *coord_var coord_var coord_var …*

This sets one or more coordinate variables.
coord_var can be a name, OOC-name or `""`.
If *coord_var* is a valid name then this is also used as dimension name if this is undefined.
If *coord_var* is `""` then any existing coordinate variable is removed.
If the number of *coord_var* arguments < rank then trailing values default to `""`. Thus the following command removes all coordinate variables:

`ooc_name` `se coo`

See above ("**Method** `coordinate`") for an example.

`set count`

ooc_name `set count` ?*int*? ?`-keep`?

This sets or increments the reference count. One situation where this facility is needed is where a GUI window points to a NAO and must be retained until the window is destroyed.
`-keep`: Do not delete NAO (with new count = 0)

`-keep`: Do not delete NAO with reference count of 0 If *int* is signed then add it to reference count.
If *int* is unsigned then set reference count to *int*.
If *int* not specified then add 1 to reference count (i.e. treat as "+1")

`Set dimension`

ooc_name `set dimension` *dim_name dim_name dim_name …*

This sets one or more dimension names.
If *dim_name* is a tcl name pointing to a NAO then this also defines the coordinate variable if this is undefined.
If *dim_name* is `""` then any existing dimension name is removed.
If the number of *dim_name*s < rank then trailing values default to `""`. Thus the following command removes all dimension names:

`ooc_name` `se d`

Example:

```
% $x set dim time
% $x dim
```

`set label`

ooc_name `set label` ?*string*?

This sets the label (title). The default is NULL i.e. no label.

See above ("**Method** `label`") for an example.

`Set link`

ooc_name `set link` ?*nao*?

This sets the link slot number to point to a NAO
The default is NULL i.e. no link.

See above ("**Method** `link`") for an example.

`Set missing`

ooc_name `set missing` ?*value*?

This sets the missing value. The default is NULL i.e. no missing value.

Example:

```
% $x set miss 0
% $x
_ 0.5 1 1.5 2
```

Note that the value of 0 is now treated as missing.

`Set unit`

ooc_name `set unit` ?*unit*?

This sets the unit of measure. The default is NULL i.e. no unit.

See above ("**Method** `unit`") for an example.

`Set value`

ooc_name `set value` ?*value*? ?*index*?

This sets the value of data elements selected by NAO *index* (default: `""` i.e. whole array) to new values copied from successive elements of NAO *value* (default: `""` meaning null value). Elements of *value* are recycled if necessary.

Methods which Write to File or Internal Structure

Method `blt_vector`

ooc_name `blt_vector` *name*

This sets the BLT vector named *name* (which may or may not already exist) to the value of the data in the NAO.

Method `hdf`

ooc_name `hdf` ?*switches*? *filename sds*

This writes data from the NAO to an SDS named *sds* within an HDF file named *filename*.

switches can be:
`-unlimited`: Create sds with unlimited dimension 0
`-coordinateVariable` *expr*: boxed NAO which specifies coordinate variables.

`-datatype` *type*: HDF datatype: `c8, i8, i16, i32, u8, u16, u32, f32` or `f64`
`-range` *float*: HDF `valid_range`
`-scale` *float*: HDF `scale_factor`
`-offset` *float*: HDF `add_offset`
`-index` *expr*: position within SDS where data is to be written.

If "`-coordinateVariable` *expr*" is not specified then the coordinate variables of the main NAO are used if these exist.

If "`-index` *expr*" is not specified then the coordinate variables of the main NAO are used if these exist, otherwise writing starts at the beginning of the SDS.

Example:

```
% [nap "3 .. 7"] hdf simple.hdf vec
% exec hdp dumpsds simple.hdf
File name: simple.hdf

Variable Name = vec
         Index = 0
         Type= 32-bit signed integer
         Ref. = 2
         Rank = 1
         Number of attributes = 1
         Dim0: Name=fakeDim0
                 Size = 5
                 Scale Type = number-type not set
                 Number of attributes = 0
         Attr0: Name = _FillValue
                 Type = 32-bit signed integer
                 Count= 1
                 Value = -2147483648
         Data :
                3 4 5 6 7
```

Method `netcdf`

ooc_name `netcdf` ?*switches*? *filename var*

This writes data from the NAO to a netCDF variable named *var* within the netCDF file named *filename*.

switches can be:
`-unlimited`: Create variable with unlimited dimension 0
`-coordinateVariable` *expr*: boxed NAO which specifies coordinate variables.
`-datatype` *type*: netCDF datatype: `c8, i16, i32, u8, f32` or `f64`
`-range` *float*: netCDF `valid_range`

`-scale` *float*: netCDF `scale_factor`
`-offset` *float*: netCDF `add_offset`
`-index` *expr*: position within netCDF variable where data is to be written.

If "`-coordinateVariable` *expr*" is not specified then the coordinate variables of the main NAO are used if these exist.

If "`-index` *expr*" is not specified then the coordinate variables of the main NAO are used if these exist, otherwise writing starts at the beginning of the netCDF variable.

Example:

```
% $y net sq.nc area -coordinateVariable "x // {3 6 9}"
::NAP::241-241
% exec ncdump sq.nc
netcdf sq {
dimensions:
        x = 8 ;
variables:
        double x(x) ;
        double area(x) ;
                area:_FillValue = nan ;
                area:long_name = "areas of squares" ;
                area:units = "seconds" ;
data:

 x = 0, 0.5, 1, 1.5, 2, 3, 6, 9 ;

 area = 0, 0.25, 1, 2.25, 4, nan, nan, nan ;
}
```

Note that the netCDF variable `area` is dimensioned to 8 (the shape of the argument specified by the `-coordinateVariable` option).

Method `write`

ooc_name `write` ?*tcl_channel*?

This writes raw (binary) data to the file specified by *tcl_channel*, which defaults to `stdout` (standard output). For example:

```
% set file [open y.tmp w]
file4
% $y write $file
% close $file
% set file [open y.tmp]
```

```
file4
% [nap_get binary $file f64] all
::NAP::248-248  f64  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 5      Name: (NULL)    Coordinate-variable: (NULL)
Value:
0 0.25 1 2.25 4
% close $file
```

NAP Constants

Nap provides a rich variety of constants. Nap is oriented to numeric data but does provide string constants. The data-type can be specified as a suffix (except for strings and hexadecimal constants). Numeric constants can be scalars (simple numbers) or higher-rank arrays.

Integer Scalar Constants

An integer scalar constant can be specified in decimal, octal or hexadecimal form. Octal and hexadecimal integer constants begin with a zero.

The default data-type is `i32` (32-bit signed integer) for decimal integer constants and `u32` (32-bit unsigned integer) for octal integer constants. A data-type suffix is not allowed for hexadecimal constants because some cases would be ambiguous. Hexadecimal constants are 32-bit unsigned integers.

Examples of integer constants are:

```
% [nap 14] all
::NAP::72-72  i32  MissingValue: -2147483648  References: 0  Unit: (NULL)
Value:
14
% [nap 14u8] all
::NAP::74-74  u8  MissingValue: (NULL)  References: 0  Unit: (NULL)
Value:
14
% [nap 014] all
::NAP::76-76  u32  MissingValue: 4294967295  References: 0  Unit: (NULL)
Value:
12
% [nap 014i8] all
::NAP::78-78  i8  MissingValue: -128  References: 0  Unit: (NULL)
Value:
12
% [nap 0x14] all
::NAP::80-80  u32  MissingValue: 4294967295  References: 0  Unit: (NULL)
Value:
20
```

The constant "`_`" represents an `i32` NAO whose value and missing-value are both -2147483648 (the minimum possible `i32` value). It provides a convenient way of indicating undefined data. Such values are used mainly within array constants and will be discussed further in that section.

Floating-point Scalar Constants

A floating-point scalar constant can represent ∞, NaN or a normal finite value. A finite value is represented by a mantissa, optionally followed by an exponent. There can be a data-type suffix on any floating-point scalar constant. If this suffix is omitted the data-type is `f64` (64-bit float).

A mantissa can be written in either decimal or rational form. A decimal mantissa must not begin or end with a decimal point. A rational mantissa consists of two integers separated by `r` and represents their ratio. Here are examples of floating-point constants without exponents:

```
% [nap 4.0] all
::NAP::82-82  f64  MissingValue: NaN  References: 0  Unit: (NULL)
Value:
4
% [nap 4f32] all
::NAP::83-83  f32  MissingValue: NaN  References: 0  Unit: (NULL)
Value:
4
% [nap 2r3] all
::NAP::85-85  f64  MissingValue: NaN  References: 0  Unit: (NULL)
Value:
0.666667
```

The letter `e` indicates an exponent with base 10. The letter `p` indicates an exponent with base π. Examples of constants with exponents are:

```
% [nap 1e4] all
::NAP::89-89  f64  MissingValue: NaN  References: 0  Unit: (NULL)
Value:
10000
% [nap 1p1]
3.14159
% [nap 180p-1]; # degrees in a radian
57.2958
% [nap 1r3p1f32] all; # pi/3
::NAP::95-95  f32  MissingValue: NaN  References: 0  Unit: (NULL)
Value:
1.0472
```

Infinity is represented by `1i`. NaN is represented by `1n`. Examples are:

```
% [nap 1i] all
::NAP::101-101  f64  MissingValue: NaN  References: 0  Unit: (NULL)
Value:
Inf
% [nap 1if32] all
::NAP::102-102  f32  MissingValue: NaN  References: 0  Unit: (NULL)
```

```
Value:
Inf
% [nap 1n] all
::NAP::104-104  f64  MissingValue: NaN  References: 0  Unit: (NULL)
Value:
_
% [nap 1nf32] all
::NAP::105-105  f32  MissingValue: NaN  References: 0  Unit: (NULL)
Value:
_
```

Numeric Array Constants

Tcl uses nested braces ("`{}`") to represent lists. Nap uses braces in a similar manner to represent n-dimensional constant arrays. The elements of array constants have the same form as scalar constants.

A vector (1-dimensional array) constant is enclosed by one level of braces, as in:

```
% [nap "{2 -7 8}"] all
::NAP::110-110  i32  MissingValue: -2147483648  References: 0  Unit: (NULL)
Dimension 0   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
2 -7 8
```

A matrix (2-dimensional array) constant is enclosed by two levels of braces, as in:

```
% [nap "{{1 3 5}{2 4 6}}"] all
::NAP::120-120  i32  MissingValue: -2147483648  References: 0  Unit: (NULL)
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
1 3 5
2 4 6
```

We could have written each row on a separate line, as in

```
% [nap "{
    {1 3 5}
    {2 4 6}
}"]
1 3 5
2 4 6
```

The following generates a three-dimensional constant:

```
% [nap "{{{1 5 0}{2 2 9}}{{3 0 7}{4 4 9}}}"] all
::NAP::126-126  i32  MissingValue: -2147483648  References: 0  Unit: (NULL)
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 2   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
1 5 0
2 2 9

3 0 7
4 4 9
```

Negative elements are preceded by "`-`". Repeated elements and sub-arrays can be specified using "#" which also has a related meaning as an operator. The following illustrates such repetition counts:

```
% [nap "{{7 3#5} 2#{9 1 2#4}}"] all
::NAP::131-131  i32  MissingValue: -2147483648  References: 0  Unit: (NULL)
Dimension 0   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 4      Name: (NULL)    Coordinate-variable: (NULL)
Value:
7 5 5 5
9 1 4 4
9 1 4 4
```

Undefined (missing) elements are represented by "`_`", as in:

```
% [nap "{1.6 _ 0}"] all
::NAP::133-133  f64  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
1.6 _ 0
```

It is possible to include data-type suffices on individual elements, but it is more convenient to use a data conversion function to obtain the desired data-type. For example:

```
% [nap "f32{0 -6 1e9 1p1}"] all
::NAP::137-137  f32  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 4      Name: (NULL)    Coordinate-variable: (NULL)

Value:
0 -6 1e+09 3.14159
```

String Constants

String constants are enclosed by either two apostrophes ("`'`") or two grave accents ("`` ` ``"). String constants have the data-type `c8` (8-bit character). They are 1-dimensional (vectors) but other ranks can be produced using the function `reshape`. A simple string constant is shown by:

```
% [nap "'Hello world'"] all
::NAP::139-139  c8  MissingValue: (NULL)  References: 0  Unit: (NULL)
Dimension 0   Size: 11     Name: (NULL)    Coordinate-variable: (NULL)
Value:
Hello world
```

Adjacent strings are concatenated as in:

```
% [nap "`can't` ' go'"]
can't go
```

NAP Operators

Operators and Precedence

The following table is essentially a superset of **Table 5.2** in Ousterhout's 1994 classic *Tcl and the Tk Toolkit*. As there, groups of operators between horizontal lines have the same precedence; higher groups have higher precedence.

Operators are left-associative unless specified otherwise. For example, `**` is right-associative, as shown by:

```
% [nap "10 ** 2 ** 3"]
1e+08
```

The nature of arguments is indicated as follows:
a and *b* represent general arrays.
x and *y* represent scalars.
u and *v* represent vectors.
A and *B* represent matrices.
n represents a Tcl name, which may include namespaces.
p represents a boxed vector of pointers to arrays a_0, a_1, a_2, …

'AP' means *arithmetic progression*.

Syntax	Result
a`**`*b*	a^b. Right-associative
`+`*a*	New copy of *a*
`-`*a*	Negative of *a*
`!`*a*	Logical NOT: 1 if *a* is zero, else 0
`~`*a*	Bit-wise complement of *a*
`#`*a*	Frequencies of values 0, 1, 2, …
`@`*a*	Indirect subscript
`@@`*a*	Indirect subscript

`v@b`	s such that $v_s=b$, where v is ordered vector
`v@@b`	`i32` s for which $\lvert v_s-b \rvert$ is least
`v@@@b`	smallest `i32` s for which $v_s=b$
`[a]...[b]`	Boxed vector pointing to a and b
`x..y`	AP from x to y in steps of `+1` or `-1`
`x..p`	AP from x to a_0 in steps of a_1
`p..y`	AP from a_1 to y with a_0 elements
`u#v`	u copies of v
`p#b`	Cross-product replication
`u+*v`	(u and v vectors) Scalar (dot) product
`A+*B`	(A and B matrices) Matrix product
`a*b`	$a \times b$
`a/b`	$a \div b$
`a%b`	Remainder after dividing a by b
`a+b`	$a + b$
`a-b`	$a - b$
`a<<b`	Left-shift a by b bits
`a>>b`	Right-shift a by b bits
`a<<<b`	Lesser of a and b
`a>>>b`	Greater of a and b
`a<b`	1 if $a < b$, else 0
`a>b`	1 if $a > b$, else 0
`a<=b`	1 if $a \le b$, else 0
`a>=b`	1 if $a \ge b$, else 0
`a==b`	1 if $a = b$, else 0

`a!=b`	1 if $a \neq b$, else 0
a&*b*	Bit-wise AND of *a* and *b*
`a^b`	Bit-wise exclusive OR of *a* and *b*
`a|b`	Bit-wise OR of *a* and *b*
a&&*b*	Logical AND: 1 if $a \neq 0$ and $b \neq 0$, else 0
`a||b`	Logical OR: 1 if $a \neq 0$ or $b \neq 0$, else 0
`a?b:c`	Choice: if $a \neq 0$ then *b*, else *c*
`a//b` `a///b`	Concatenate along existing dimension Concatenate along new dimension
[*a*],[*b*]	Boxed vector pointing to *a* and *b*
n=*a*	Result is *a*. Right-associative Side Effect: Set *n* to OOC-name of *a*

Assignment Operator "="

The "`nap`" command (unlike "`expr`") allows the assignment operator "`=`". The left-hand operand must be a Tcl name, which is used to define a Tcl variable whose (string) value is set to the OOC-name of the right-hand operand. The assignment operator has a result like any other operator. This result is the value of the right-hand operand. This is shown in the following:

```
% nap "a = (b = 6) + 2"
::NAP::15-15
% $b
6
% $a
8
```

The assignment operator has the lowest precedence and is right-associative, allowing expressions such as:

```
% nap "a = 3 + b = {1.5 0}"
::NAP::16-16
% $b
1.5 0
% $a
4.5 3
```

Link Operators "`...`" and "`,`"

The link operators "`...`" and "`,`" are identical except for precedence. They produce a boxed vector pointing to the operands.

A common use of "`,`" is to pass multiple arguments to a function. For example the logarithm function `log` takes an optional second argument specifying *base*, as in:

```
% [nap "log(32, 2)"]
5
```

The operator "`,`" is also used in *cross-product indexing*, as discussed in the section NAP Indexing.

A common use of "`...`" is in conjunction with the arithmetic progression operator "`..`", which is discussed in the next section.

The left operand of "`...`" or "`,`" generates one boxed vector and the right operand generates another. These two boxed vectors are concatenated to form the result, which is also a boxed vector. If the data-type of an operand is not boxed then it generates a single-element boxed vector pointing to it. If an operand is a boxed vector then it generates a copy of itself. If an operand is a boxed scalar then it is treated as a boxed vector with a single element. If an operand is absent (NULL) then it generates a single-element (whose value is 0, the missing-value) boxed vector.

Arithmetic Progression Operator "`..`"

The operator "`..`" generates an arithmetic progression. If both operands are simple numeric scalars then the step size is +1 or −1, the left operand specifies the first value and the right operand specifies the final value. For example:

```
% [nap "3 .. 6"]
3 4 5 6
```

```
% [nap "6 .. 3"]
6 5 4 3
% [nap "1.8 .. -1.2"]
1.8 0.8 -0.2 -1.2
```

If the difference between the operands is not an integral multiple of the step size then the final step is smaller than the preceding steps. This is shown by:

```
% [nap "2.3 .. 5.9"]
2.3 3.3 4.3 5.3 5.9
```

The right operand can be a boxed two-element vector pointing to the final value and the step size. Such a boxed operand is usually generated using the operator "`...`", as in:

```
% [nap "3 .. 9 ... 2"]
3 5 7 9
% [nap "0 .. -1.6 ... -0.5"]
0 -0.5 -1 -1.5 -1.6
```

The left operand can be a boxed two-element vector pointing to the number of elements and the first value. Such a boxed operand is also usually generated using the operator "`...`", as in:

```
% [nap "5 ... 1 .. 7"]
1 2.5 4 5.5 7
```

It is not legal for both operands to be boxed. It is legal to specify a non-integral number of elements, as in:

```
% [nap "3.5 ... 2 .. 12"]
2 6 10 12
```

Note that 3.5 elements means 2.5 steps. There are two full steps of 4, followed by a half step of 2. When the left operand is boxed the step size is calculated using $(final - first)/(n - 1)$, where n is the number of elements.

The data-type of the result depends on the data-types of *first*, *final* and *step*. For example:

```
% [nap "1 .. 7.0 ... 2"] all
::NAP::262-262  f64  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 4      Name: (NULL)    Coordinate-variable: (NULL)
Value:
1 3 5 7
```

Concatenation Operators "//" and "///"

The following example illustrates the difference between "//" and "///":

```
% [nap "{5 2} // {9 8}"]
5 2 9 8
% [nap "{5 2} /// {9 8}"]
5 2
9 8
```

However if the arguments differ in rank (by 1) then both operators are equivalent, as shown by:

```
% [nap "{{1 3 5}{2 4 6}} // {-1 -9 -2}"]
 1  3  5
 2  4  6
-1 -9 -2
% [nap "{{1 3 5}{2 4 6}} /// {-1 -9 -2}"]
 1  3  5
 2  4  6
-1 -9 -2
```

This is convenient for expressions such as:

```
% [nap "{5 2} /// {9 8} /// {0 1}"]
5 2
9 8
0 1
```

Inverse Indexing Operators "@", "@@" and "@@@"

These three operators all take an optional vector left argument. The result is a subscript of this vector. The left argument defaults to the coordinate-variable of the dimension (only relevant to indirect indexing).

Interpolated Subscript "@"

The "@" operator requires a sorted left argument. The result of *v*@*b* is the `f32` subscript *s* such that *v*(*s*)==*b*. For example:

```
% [nap "{1.5 3.4 3.6 4} @ 3.5"]
1.5
% [nap "{1.5 3.4 3.6 4} @ 3.7"]
2.25
```

Note that 3.5 is halfway between 3.4 (subscript 1) and 3.6 (subscript 2), so the first result is 1.5. Similarly, 3.7 is quarterway between 3.6 (subscript 2) and 4 (subscript 3), so the second result is 2.25.

Combining these two examples into one:

```
% [nap "{1.5 3.4 3.6 4} @ {3.5 3.7}"]
1.5 2.25
```

We can check this result by using it as an index:

```
% [nap "{1.5 3.4 3.6 4} {1.5 2.25}"]
3.5 3.7
```

Subscript of Closest "@@"

The result of *v*@@*b* is the `i32` subscript *s* for which `abs(`*v*`(`*s*`)-`*b*`)` is least. For example:

```
% [nap "{1.5 3.4 0 2.4 -1 0} @@ {2 -99}"]
3 4
```

Element 3 has the value 2.4, which is the closest to 2. Element 4 has the value -1, which is the closest to -99.

Subscript of Match "@@@"

The result of *v*@@@*b* is the smallest `i32` subscript *s* for which *v*`(`*s*`)==`*b*. For example:

```
% [nap "{3 2 9 2 0 3} @@@ {0 3 2}"]
4 0 1
```

Element 4 is the only 0, element 0 is the first 3 and element 1 is the first 2.

The following example shows that this operator can be used with character data:

```
% [nap 'hello world' @@@ 'wol']
6 4 2
```

Tally Unary Operator "#"

Unary "#" produces a frequency table. It tallies the number of 0s, 1s, 2s, …, as in the following:

```
% [nap "#{2 5 4 5 2 -3 0 2}"]
1 0 3 0 1 2
```

There is one zero, no ones, three twos, no threes, one four and two fives. Note that the negative value (-3) is ignored.

If the argument has more than 1 dimension then the result has the same shape, except that the size of the first dimension is changed to *m*+1, where *m* is the maximum value. Each element of the result is a frequency tallied over the first dimension. For example:

```
% [nap "{{2 5 4 5}{2 -3 0 2}}"]
 2  5  4  5
 2 -3  0  2
% [nap "#{{2 5 4 5}{2 -3 0 2}}"]
0 0 1 0
0 0 0 0
2 0 0 1
0 0 0 0
0 0 1 0
0 1 0 1
```

Replicate Binary Operator "#"

can appear within array constants, as in:

```
% [nap "{7 3#8 0}"]
7 8 8 8 0
```

The # operator has a related meaning, as shown by:

```
% [nap "3#8"]
8 8 8
% [nap "{4 1 0 2} # {7 12 9 8}"] value
7 7 7 7 12 8 8
```

Each element of the left argument defines the number of replications of the corresponding element of the right argument. The arguments can be vectors or scalars. The result is a vector.

Note that one can use this operator to select from a vector those elements which satisfy some condition. The following example selects the even elements:

```
% nap "x = {9 1 0 2 3 -8 0}"
::NAP::286-286
% [nap "(x % 2 == 0) # x"]
0 2 -8 0
```

This works because the left-hand argument is:

```
% [nap "(x % 2 == 0)"] value
0 0 1 1 0 1 1
```

If the right-hand argument *b* is multidimensional then the left-hand argument must be a boxed vector pointing to a vector corresponding to each dimension of *b*. For example:

```
% nap "mat = reshape(1 .. 12, {3 4})"
::NAP::316-316
% $mat
 1  2  3  4
 5  6  7  8
 9 10 11 12
% [nap "({2 0 1},{3 2 0 1}) # mat"]
 1  1  1  2  2  4
 1  1  1  2  2  4
 9  9  9 10 10 12
```

This is equivalent to using the following cross-product index:

```
% [nap "mat({0 0 2},{0 0 0 1 1 3})"]
 1  1  1  2  2  4
 1  1  1  2  2  4
 9  9  9 10 10 12
```

Remainder Operator "%"

The value of the remainder $r = a \% b$ is defined for all real a and b so that:
if $b > 0$ then $0 \le r < b$
if $b = 0$ then $r = 0$

if $b < 0$ then $b < r \leq 0$
if $a \geq 0$ and $b = \infty$ then $r = a$
if $a \leq 0$ and $b = -\infty$ then $r = a$
if $a < 0$ and $b = \infty$ then $r = \infty$
if $a > 0$ and $b = -\infty$ then $r = -\infty$.
Thus:

```
% [nap "0.7 % {0.3 0 -0.3}"]
0.1 0 -0.2
% [nap "{7 0 -7} % 1if32"]
7 0 Inf
% [nap "{7 0 -7} % -1if32"]
-Inf 0 -7
```

Indexing

Indexing is the process of extracting elements from arrays. NAP extends this concept to the estimation (using interpolation) of values *between* the elements.

An index can appear:

- within a NAP expression
- as an argument of an OOC. E.g. method `set value` takes an an argument which specifies which elements are to be modified
- as an argument of commands `nap_get hdf` and `nap_get netcdf`, specifying positions within a file

NAP provides powerful indexing (subscripting) facilities. The subscript origin is 0 (as in other aspects of Tcl such as lists). The rightmost dimension is the least significant (varies fastest). Here is a simple example of a vector indexed by a scalar:

```
% nap "vector = {2 -5 9 4}"
::NAP::14-14
% [nap "vector(2)"]
9
```

Indexing Syntax

NAP syntax specifies that indexing is implied by two adjacent NAOs, with the base array on the left and the index on the right. Thus it is not necessary to parenthesize an index which is simply a constant or variable-name. However parentheses may make the code clearer to humans, who are likely to be familiar with languages where this is required.

This syntax means that the above example can be rewritten without parentheses as:

```
% [nap "vector 2"]
9
```

It also means that any non-scalar expression (including a constant of course) can be indexed, as shown by:

```
% [nap "{2 -5 9 4} 2"]
9
% [nap "({2 -5 9 4} + 10) 2"]
19
```

Dimension-Position

A *dimension-position* is a scalar value defining the position along a dimension. Fractional values are valid and represent positions *between* the array elements. Values at non-integral positions are estimated using n-dimensional linear interpolation. The following demonstrates this (continuing the above example):

```
% [nap "vector 2.5"]
6.5
```

Note that the dimension-position `2.5` is halfway between `2` (corresponding to the value `9`) and `3` (corresponding to the value `4`). Thus the value is estimated to be
0.5 * 9.0 + 0.5 * 4.0 = 4.5 + 2.0 = 6.5
using ordinary one-dimensional linear interpolation.

If s is the dimension-size and p the position, then:
$0 \le p < s$
Values between s-1 and s are defined by treating position s as equivalent to 0. This gives wraparound useful with cyclic dimensions such as longitude. Thus

```
% [nap "vector 3.1"]
3.8
```

Note that the dimension-position `3.1` is 10% of the distance between `3` (corresponding to the value `4`) and `4` (equivalent to `0` and corresponding to the value `2`). Thus the value is estimated to be
0.9 * 4.0 + 0.1 * 2.0 = 3.6 + 0.2 = 3.8

Subscript

A *subscript* is similar to a *dimension-position* except that there are no size limits. The corresponding *dimension-position* is defined by
subscript % *s*
where *s* is the dimension-size. Note that dimension-positions can only be defined via subscripts.

Thus in our example subscript `6` is treated as `6%4` = `2`. So we get

```
% [nap "vector 6"]
9
```

It also means that negative values can be use to index backward from the end, as shown by:

```
% [nap "vector(-1)"]
4
% [nap "vector(-2)"]
9
% [nap "vector(-3)"]
-5
```

Elemental Index

An *elemental index* is a vector of *rank* subscripts, specifying the subscripts of an element of an array. The following example creates a matrix `mat` and illustrates the use of elemental indices to extract individual elements.

```
% nap "mat = {{1.5 0 7}{2 -4 -9}}"
::NAP::60-60
% $mat
 1.5  0.0  7.0
 2.0 -4.0 -9.0
```

```
% [nap "mat {0 1}"]
0
% [nap "mat {1 -1}"]
-9
%
% [nap "mat {0.5 1.5}"]
-1.5
```

The value corresponding to the index `{0.5 1.5}` is estimated, using bilinear interpolation, to be
0.25 * 0.0 + 0.25 * 7.0 + 0.25 * (-4.0) + 0.25 * (-9.0) = -1.5

Index

An *index* is an array defining one or more elemental indices. The following table lists the three types, which are explained in the sections below:

Index Type	Rank of Indexed Array
shape-preserving	1
full	2 or more
cross-product	2 or more

Shape-Preserving

Shape-preserving indexing is used to index a vector. The shape of the result is the same as that of the index. The following example shows how the previously defined variable `vector` can be indexed by

- a scalar to produce a scalar
- a vector to produce a vector
- a matrix to produce a matrix:

```
% $vector
2 -5 9 4
% [nap "vector(2)"] all
::NAP::57-57  i32  MissingValue: -2147483648  References: 0  Unit: (NULL)
Value:
9
% [nap "vector({2 2.5 2})"] all
::NAP::61-61  f32  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
9 6.5 9
% [nap "vector({
{1 0 2.5}
{-1 2 1}
})"] all

::NAP::67-67  f32  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
```

```
Value:
-5.0  2.0  6.5
 4.0  9.0 -5.0
```

The *shape-preserving* property means one can use a vector to define a mapping. The following example maps 0 to 4, 1 to 1, 2 to 9 and 3 to 4:

```
% [nap "{4 1 9 4} {
{2 1 2 0}
{3 3 0 1}
}"]
9 1 9 4
4 4 4 1
```

The following example uses the same technique to implement a simple substitution cipher (mapping space to R, A to X, B to B, C to T, … as shown) to encrypt the message "HELLO WORLD" as "A HHVREVZHC" which is then decrypted.

```
% nap "plain   = ' ABCDEFGHIJKLMNOPQRSTUVWXYZ'"
::NAP::63-63
% nap "cipher  = 'RXBTC MUAFGWHYIVJKZDLNOEPQS'"
::NAP::64-64
% [nap "plain((plain @@ cipher)(plain @@ 'HELLO WORLD'))"]; # encrypt
A HHVREVZHC
% [nap "cipher((cipher @@ plain)(cipher @@ 'A HHVREVZHC'))"]; # decrypt
HELLO WORLD
```

Full-index

A *full-index* is an array specifying a separate elemental index for every element of the result. The shape of the index is the shape of the result with *r* (the rank of the indexed array) appended. Each row of the index contains a vector of *r* elements defining an elemental index.

The following example shows how the previously defined variable `mat` can be indexed by

- a vector to produce a scalar
- a matrix to produce a vector

```
% $mat
 1.5  0.0  7.0
 2.0 -4.0 -9.0
% [nap "mat {0.5 1.5}"] all
::NAP::148-148  f32  MissingValue: NaN  References: 0  Unit: (NULL)
Value:
-1.5
% [nap "mat {
{0.5 1.5}
{0 1}
{-1 -1}
}"] all
::NAP::157-157  f32  MissingValue: NaN  References: 0  Unit: (NULL)
```

```
Dimension 0   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
-1.5 0 -9
```

Note that *shape-preserving* indexing is similar to applying *full* indexing to a vector (if this were allowed). The shape-preserving-index is the hypothetical full-index reshaped to omit the final redundant dimension of size 1.

Cross-product-index

A *cross-product-index* is a boxed vector pointing to scalars, vectors and nulls, whose cross-product combination defines the elemental indices of the indexed array. The shape of a cross-product-index is a single-element vector containing the rank of the indexed array.

A cross-product-index is usually defined using the operator "`,`". This allows the left and/or right operand to be omitted and such *null* (missing) operands are treated as `0..`*s*`-1`, where *s* is the dimension-size. Scalar operands produce no corresponding dimension in the result.

The following examples again use the previously defined variable `mat`. We begin by repeating the first *full-index* example above and then we provide the *cross-product-index* equivalent:

```
% $mat
 1.5  0.0  7.0
 2.0 -4.0 -9.0
% [nap "mat({0.5 1.5})"] all
::NAP::196-196  f32  MissingValue: NaN  References: 0  Unit: (NULL)
Value:
-1.5
% [nap "mat(0.5,1.5)"] all
::NAP::204-204  f32  MissingValue: NaN  References: 0  Unit: (NULL)
Value:
-1.5
```

The next example shows how the previously defined variable `mat` can be indexed by the cross-product of two vectors to produce a matrix, then provides the equivalent *full-index*:

```
% $mat
 1.5  0.0  7.0
 2.0 -4.0 -9.0
% [nap "mat({1 0},{2 0 -1 0})"] all
::NAP::174-174  f64  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 4      Name: (NULL)    Coordinate-variable: (NULL)
Value:
-9.0  2.0 -9.0  2.0
 7.0  1.5  7.0  1.5
% [nap "mat({
{{1 2}{1 0}{1 -1}{1 0}}
{{0 2}{2 0}{2 -1}{2 0}}
})"] all
```

```
::NAP::180-180  f64  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 4      Name: (NULL)    Coordinate-variable: (NULL)
Value:
-9.0  2.0 -9.0  2.0
 7.0  1.5  7.0  1.5
```

The following example illustrates the effect of a null operand to "`,`". It also shows the difference between a scalar operand and a single-element vector containing the same value.

```
% $mat
 1.5  0.0  7.0
 2.0 -4.0 -9.0
% [nap "mat(1,)"] all
::NAP::209-209  f64  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
2 -4 -9
% [nap "mat({1},)"] all
::NAP::213-213  f64  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 1      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
 2 -4 -9
```

The following example creates a rank-3 array `a3d` with shape `{2 2 3}`, then extracts all of row 0 from both layers:

```
% nap "a3d = {
{
{9 1 4}
{0 8 7}
}{
{2 3 5}
{9 6 0}
}
}"
::NAP::215-215
% $a3d all
::NAP::215-215  i32  MissingValue: -2147483648  References: 1  Unit: (NULL)
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 2   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
9 1 4
0 8 7

2 3 5
9 6 0
% [nap "a3d(,0,)"] all
::NAP::220-220  i32  MissingValue: -2147483648  References: 0  Unit: (NULL)
```

```
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
9 1 4
2 3 5
```

Indirect Indexing

It is often more natural to index via coordinate variables rather than subscripts. For example, consider a matrix with latitude and longitude coordinate variables. One could specify an element directly using subscripts such as "row 3, column 5". One could also specify an interpolated point directly using subscripts such as "row 3.5, column 5.2". However many users would prefer to specify latitude and longitude (values of the coordinate variables) rather than specify row and column. *Indirect indexing* simplifies such indexing via coordinate variables. The following three sections correspond to the three types of index: *shape-preserving*, *cross-product* and *full*.

Indirect Shape-Preserving Indexing

The following table defines *indirect shape-preserving* indexing of any vector *v* via any array *c* containing coordinate variable values:

Syntax	Value
v(@*c*)	*v*(coordinate_variable(*v*)@*c*)
v(@@*c*)	*v*(coordinate_variable(*v*)@@*c*)

For example, suppose we have temperatures at two-hourly intervals from time 1000 to 1600 as follows:

```
% nap "t = {20.2 21.6 24.9 22.7}"
::NAP::159-159
% $t set coord "10 .. 16 ... 2"
```

We could estimate temperatures every hour during this period as follows:

```
% [nap "t(coordinate_variable(t)@(10..16))"] value
20.2 20.9 21.6 23.25 24.9 23.8 22.7
```

Indirect indexing allows us to omit the left argument of operators "@" and "@@" in such expressions. This enables the above expression to be simplified as follows:

```
% [nap "t(@(10..16))"] value
20.2 20.9 21.6 23.25 24.9 23.8 22.7
```

Note that this syntax does not allow indirect indices such as that in "`t(3+@@12)`". Instead we have to use *binary* "@@" as in "`t(3+coordinate_variable(t)@@12)`".

Indirect Cross-Product Indexing

The syntax for a general *cross-product-index* (involving direct and/or indirect indexing) is:

[@[@]]*expr*, [@[@]]*expr*, [@[@]]*expr*, …
where *expr* is an expression (which may need to be enclosed in parentheses).

The following table defines *indirect cross-product* indexing. It shows how the subscript for dimension *d* of array *a* is calculated from (vector or scalar) *v*:

Syntax	Subscript Value
@*v*	`coordinate_variable(`*a*`,`*d*`)`@*v*
@@*v*	`coordinate_variable(`*a*`,`*d*`)`@@*v*

The following creates a 3×4 matrix `temperature` which will be used to demonstrate *indirect indexing*. It has

- unit of `degC` (degrees Celsius).
- rows corresponding to latitudes 10°N, 20°N and 30°N
- columns corresponding to longitudes 110°E, 120°E, 130°E and 140°E

```
% nap "temperature = f32{
{31.5 37.2 32.9 34.0}
{25.1 25.2 29.0 21.9}
{20.5 21.2 21.0 19.9}
}"
::NAP::72-72
% $temperature set unit degC
% nap "latitude = f32{10 20 30}"
::NAP::76-76
% $latitude set unit degrees_north
% nap "longitude = f32(110 .. 140 ... 10)"
::NAP::86-86
% $longitude set unit degrees_east
% $temperature set coo latitude longitude
```

The following verifies that the main NAO and its coordinate variables are as expected:

```
% $temperature all
::NAP::72-72  f32  MissingValue: NaN  References: 1  Unit: degC
Dimension 0   Size: 3      Name: latitude  Coordinate-variable: ::NAP::76-76
Dimension 1   Size: 4      Name: longitude  Coordinate-variable: ::NAP::86-86
Value:
31.5 37.2 32.9 34.0
25.1 25.2 29.0 21.9
20.5 21.2 21.0 19.9
% ::NAP::76-76 all
::NAP::76-76  f32  MissingValue: NaN  References: 2  Unit: degrees_north
Dimension 0   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
10 20 30
% ::NAP::86-86 all
::NAP::86-86  f32  MissingValue: NaN  References: 2  Unit: degrees_east
Dimension 0   Size: 4      Name: (NULL)    Coordinate-variable: (NULL)
```

```
Value:
110 120 130 140
```

The following illustrates the use of both direct and indirect indexing to display the value of `29` in row 1 and column 2:

```
% [nap "temperature(1,2)"]
29
% [nap "temperature(@20, @130)"]; # latitude=20 longitude=130
29
% [nap "temperature(@@20, @@130)"]
29
% [nap "temperature(1, @130)"]
29
```

In this case there is a point exactly corresponding to 20°S, 130°E, so the operators `@` and `@@` give the same result. Let us try the point 21°S, 138°E, which is not a grid point:

```
% [nap "temperature(@21, @138)"]
23
% [nap "temperature(@@21, @@138)"]
21.9
```

Now we get different results for the two operators. Operator `@` gives a value estimated using bilinear interpolation. Operator `@@` gives the data value at the nearest row (1) and column (3).

If the unary operators `@` and `@@` did not exist we would have to use their binary equivalents as follows:

```
% nap "interpolated_row = coordinate_variable(temperature,0) @ 21"
::NAP::96-96
% $interpolated_row
1.1
% nap "interpolated_col = coordinate_variable(temperature,1) @ 138"
::NAP::103-103
% $interpolated_col
2.8
% [nap "temperature(interpolated_row, interpolated_col)"]
23
% nap "nearest_row = coordinate_variable(temperature,0) @@ 21"
::NAP::112-112
% $nearest_row
1
% nap "nearest_col = coordinate_variable(temperature,1) @@ 138"
::NAP::119-119
% $nearest_col
3
% [nap "temperature(nearest_row, nearest_col)"]
21.9
```

Say we want to estimate temperatures on a grid with

- latitudes 19°N, 20°N and 21°N
- longitudes 121°E, 122°E 123°E and 124°E

Naming the new matrix `region_temperature`, this can be done as follows:

```
% nap "region_temperature = temperature(@(19 .. 21), @(121 .. 124))"
::NAP::147-147
% $region_temperature all
::NAP::147-147  f32  MissingValue: NaN  References: 1  Unit: degC
Dimension 0   Size: 3      Name: latitude  Coordinate-variable: ::NAP::145-145
Dimension 1   Size: 4      Name: longitude  Coordinate-variable: ::NAP::146-146
Value:
26.699 26.998 27.297 27.596
25.580 25.960 26.340 26.720
25.140 25.480 25.820 26.160
% ::NAP::145-145 all
::NAP::145-145  i32  MissingValue: -2147483648  References: 1  Unit: degrees_north
Dimension 0   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
19 20 21
% ::NAP::146-146 all
::NAP::146-146  i32  MissingValue: -2147483648  References: 1  Unit: degrees_east
Dimension 0   Size: 4      Name: (NULL)    Coordinate-variable: (NULL)
Value:
121 122 123 124
```

Why has the new longitude coordinate-variable been converted to data-type `f32`? NAP recognises `degrees_east` as a special unit implying longitude characteristics such as

- wrap around to allow interpolation across longitude 180
- data-type `f32`

Indirect Full Indexing

A full index can be preceded by the unary "`@`" or "`@@`" operator to give indirect indexing. However, unlike cross-product indexing, this operator applies to all dimensions. The following example shows how full indexing can be used to produce the same values as those in the above example of cross-product indexing.

```
% [nap "temperature({{1 2}{1.1 2.8}})"]; # direct indexing
29 23
% [nap "temperature(@{{20 130}{21 138}})"]
29 23
% [nap "temperature(@@{{20 130}{21 138}})"]
29 21.9
```

How Indirect Indexing Works

Indirect indexing uses ancillary NAOs linked to index NAOs using their link slots. These ancillary NAOs contain integers with value

- 0 for direct indexing
- 1 for indirect indexing using "`@`"

• 2 for indirect indexing using "`@@`".

The unary "`@`" and "`@@`" operators simply create a copy of their operand and attach to it such an ancillary NAO. Another process in which indirect full indices are created by attaching such an ancillary NAO, is the function `invert_grid()`.

Built-in Functions

Elemental Functions

The result of an elemental function has the same shape as its argument(s). Each element of the result is defined by applying the function to the corresponding element of the argument.

The following table is very similar to **Table 5.3** in Ousterhout's 1994 classic *Tcl and the Tk Toolkit*:

Function	Result
`abs(`*x*`)`	Absolute value of x
`acos(`*x*`)`	Arc cosine of x, in the range 0 to π
`asin(`*x*`)`	Arc sine of x, in the range $-\pi/2$ to $\pi/2$
`atan(`*x*`)`	Arc tangent of x, in the range $-\pi/2$ to $\pi/2$
`atan(`*y*`,`*x*`)`	Arc tangent of y/x, in the range $-\pi$ to π
`atan2(`*y*`,`*x*`)`	Alias for `atan`
`ceil(`*x*`)`	Smallest integer not less than x
`cos(`*x*`)`	Cosine of x (x in radians)
`cosh(`*x*`)`	Hyperbolic cosine of x
`exp(`*x*`)`	e^x, where e is base of natural logarithms
`floor(`*x*`)`	Largest integer not greater than x
`fmod(`*x*`,`*y*`)`	$x\%y$
`isnan(`*x*`)`	1 if x is NaN, 0 otherwise
`log(`*x*`)`	$\log_e x$ (natural logarithm of x)
`log(`*x*,*y*`)`	$\log_y x$
`log10(`*x*`)`	$\log_{10} x$
`hypot(`*x*`,`*y*`)`	$\sqrt{}(x^2+y^2)$
`nint(`*x*`)`	Nearest integer to x
`pow(`*x*`,`*y*`)`	x^y
`random(`*x*`)`	`f32` or `f64` random number r such that $0 \le r < x$
`round(`*x*`)`	Alias for `nint`
`sign(`*x*`)`	Sign of x, i.e. $(x>0)-(x<0)$
`sin(`*x*`)`	Sine of x (x in radians)
`sqrt(`*x*`)`	$\sqrt{x}$
`sinh(`*x*`)`	Hyperbolic sine of x

`tan(`*x*`)`	Tangent of *x* (*x* in radians)
`tanh(`*x*`)`	Hyperbolic tangent of *x*

The following data-type conversion functions are also elemental:
`c8(`*x*`)`
`f32(`*x*`)`
`f64(`*x*`)`
`i8(`*x*`)`
`i16(`*x*`)`
`i32(`*x*`)`
`u8(`*x*`)`
`u16(`*x*`)`
`u32(`*x*`)`
Here are some examples of their use:

```
% [nap "f32(97 .. 102)"] all; # convert from i32 to f32
::NAP::43-43  f32  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 6      Name: (NULL)    Coordinate-variable: (NULL)
Value:
97 98 99 100 101 102
% [nap "u8('abcdef')"]; # Display ASCII codes for 'abcdef'
97 98 99 100 101 102
% [nap "c8(97 .. 102)"]; # Reverse this process
abcdef
```

Reduction and Scan Functions

A *reduction* or *insert* function is one which has the effect of inserting a binary operator between the *cells* of its argument. If the argument is a vector then its elements are the cells and the result is a scalar. If the argument is a matrix then its rows are the cells and the result is a vector containing the sum of each column. Such functions are termed *reductions* because the result has a rank which is one less than the argument.

A classic example is the ∑ summation operation, which corresponds to the NAP function `sum`. This can be used as follows to produce a scalar (rank 0) result by summing a vector (rank 1):

```
% [nap "sum({0.5 2 -1 8})"]
9.5
```

This function `sum` can be applied to a matrix (rank 2) to produce a vector (rank 1). If the second argument is omitted then we get the sum of each column. If it is 1 we get the sum of each row. This is shown by:

```
% nap "mat = {
{2 5 0}
{6 7 1}
}"
```

```
::NAP::49-49
% [nap "sum mat"]
8 12 1
% [nap "sum(mat,1)"]
7 14
```

The NAP reduction and scan functions are listed in the following table:

Function	Type	Result
`count(`*x*`[,`*r*`])`	reduction	Number of non-missing elements in rank-*r* sub-arrays of *x*
`max(`*x*`[,`*r*`])`	reduction	Maximum of rank-*r* sub-arrays of *x*
`min(`*x*`[,`*r*`])`	reduction	Minimum of rank-*r* sub-arrays of *x*
`prod(`*x*`[,`*r*`])`	reduction	Product of rank-*r* sub-arrays of *x*
`psum(`*x*`)`	scan	Partial sums of *x* (see below)
`sum(`*x*`[,`*r*`])`	reduction	Sum of rank-*r* sub-arrays of *x*

The optional second argument of reduction functions is called the *verb-rank* (as in J). It specifies the rank of the sub-arrays (cells) to which the process is applied. In the above example the verb-rank was 1, so the matrix was split into vectors (corresponding to each row) before doing the summation. This final summation process is always done by summing along the first (most significant) dimension.

It is possible to specify a verb-rank of 0. This is useful with `count()` because each (rank 0) element is processed separately. If it is missing the result is 0, otherwise it is 1. So this gives us an elemental function for testing whether values are missing. Note that the rank does not change in this case! E.g.

```
% [nap "count({4 _ 2 -9}, 0)"]
1 0 1 1
```

The result of `psum(`*x*`)` has the same shape as *x*. If *x* is a vector and *r* is the result (with the same shape) then each element of *r* is defined by

$$r_I = \sum_{i=0}^{I} x_i$$

If *x* is a matrix and *r* is the result (with the same shape) then each element of *r* is defined by

$$r_{IJ} = \sum_{i=0}^{I} \sum_{j=0}^{J} x_{ij}$$

The following example shows how partial sums can be used to calculate a 3-point moving-average in an efficient manner:

```
% nap "x = {2 7 1 3 8 2 5 0 2 5}"
```

```
::NAP::136-136
% nap "ps = 0 // psum(x)"
::NAP::141-141
% $ps all -col -1
::NAP::141-141  i32  MissingValue: -2147483648  References: 1  Unit: (NULL)
Dimension 0   Size: 11     Name: (NULL)    Coordinate-variable: (NULL)
Value:
0 2 9 10 13 21 23 28 28 30 35
% [nap "(ps(3 .. 10) - ps(0 .. 7)) / 3.0"] value
3.33333 3.66667 4 4.33333 5 2.33333 2.33333 2.33333
```

Can you allow for missing values? How about averages of a moving window in 2 dimensions?

Other similar *scan* functions can be defined for partial products and so on. However NAP currently has only `psum`.

Metadata Functions

Metadata functions return information (other than data values) from a NAO. The same information can be obtained using an OOC, but these functions are more convenient within expressions.

Function	Result
`coordinate_variable(`*x*[*d*]`)`	Coordinate variable of dimension *d* (default 0)
`label(`*x*`)`	Descriptive title
`missing_value(`*x*`)`	Value indicating null or undefined data
`nels(`*x*`)`	Number of elements = `prod(`*shape*`)`
`rank(`*x*`)`	Number of dimensions = `nels(`*shape*`)`
`shape(`*x*`)`	Vector of dimension sizes

Functions which change shape or order

Function	Result
`sort(`*x*`)`	Sort *x* into ascending order
`reshape(`*x*`)`	Spread the elements of *x* into a vector with shape `nels(`*x*`)`
`reshape(`*x*,*s*`)`	Reshape the elements of *x* into an array with shape *s*
`transpose(`*x*`)`	Reverse the order of dimensions of *x*
`transpose(`*x*,*p*`)`	Permute the dimensions of *x* to the order specified by *p*

Here are some examples of the use of these functions:

```
% [nap "sort {6.3 0.5 9 -2.1 0}"]
-2.1 0 0.5 6.3 9
```

```
% [nap "reshape {{1 3 7}{0 9 2}}"] all
::NAP::217-217  i32  MissingValue: -2147483648  References: 0  Unit: (NULL)
Dimension 0   Size: 6      Name: (NULL)    Coordinate-variable: (NULL)
Value:
1 3 7 0 9 2
% [nap "reshape({6.3 0.5 9 -2.1 0}, {2 4})"] all
::NAP::224-224  f64  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 4      Name: (NULL)    Coordinate-variable: (NULL)
Value:
 6.3  0.5  9.0 -2.1
 0.0  6.3  0.5  9.0
% [nap "transpose {{1 3 7}{0 9 2}}"] all
::NAP::228-228  i32  MissingValue: -2147483648  References: 0  Unit: (NULL)
Dimension 0   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Value:
1 0
3 9
7 2
```

Linear-algebra Functions

The function `solve_linear(`*A*[,*B*]) solves a system of linear equations defined by matrix *A* and right-hand-sides *B*. *B* can be either a vector or a matrix (representing multiple right-hand sides). If *B* is omitted then the result is the matrix inverse.

If the system is *over-determined* (more equations than unknowns) then the result is the solution of the *linear least-squares problem*. This solution minimizes the sum of the squares of the differences between the left and right-hand sides.

The following system of linear equations is solved by the following example:

$3x - 4y = 20$

$-5x + 8y = -36$

```
% nap "A = {
{3 -4}
{-5 8}
}"
::NAP::14-14
% nap "B = {20 -36}"
::NAP::17-17
% nap "x = solve_linear(A, B)"
::NAP::20-20
% $x a
::NAP::20-20  f64  MissingValue: NaN  References: 1  Unit: (NULL)
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
```

```
Value:
4 -2
```

We can check the result using matrix multiplication:

```
% [nap "A +* x"]
20 -36
```

Correlation

Function `correlation` calculates Pearson product-moment correlations (omitting cases where either value is missing). If *x* or *y* is `f64` then the result is `f64`, else it is `f32`. (But calculation is still done using `f64`.) Dimension 0 of the result has size 2. Index 0 of this dimension corresponds to the correlation values themselves, while index 1 corresponds to the sample sizes (number of non-missing data elements) used to calculate these values .

Usage:
`correlation(`x[, y, [lag_0, lag_1, …]]`)`

If the only argument is *x*, then it must be a matrix. Let *nc* be the number of columns in this matrix data argument. In this case the result has the shape 2 by *nc* by *nc*. Layer 0 contains the correlation matrix. Element r_{ij} of this matrix is the correlation between columns *i* and *j* of matrix *x*.

The following example is from Table 15.2 (page 274) of *Schaum's Outline of Theory and Problems of Statistics*, M.R. Spiegel, 1961:

```
% [nap "correlation{
        {64 57 8}
        {71 59 10}
        {53 49 6}
        {67 62 11}
        {55 51 8}
        {58 50 7}
        {77 55 10}
        {57 48 9}
        {56 52 10}
        {51 42 6}
        {76 61 12}
        {68 57 9}
}"] -f %6.4f
 1.0000  0.8196  0.7698
 0.8196  1.0000  0.7984
 0.7698  0.7984  1.0000

12.0000 12.0000 12.0000
12.0000 12.0000 12.0000
12.0000 12.0000 12.0000
```

Layer 0 of the result is the correlation matrix.
The correlation between columns 0 and 1 is 0.8196.
The correlation between columns 0 and 2 is 0.7698.
The correlation between columns 1 and 2 is 0.7984.
There is no missing data, so all values in layer 1 are 12.

If *y is* specified then the result contains one or more correlations between *x* and *y*. The ranks of *x* and *y* must be the same. (The current version supports ranks 1 and 2 only.)

If *x* and *y* have the same shape then the result contains a single correlation, calculated by treating the elements of each array as two lists of values. An example is:

```
% [nap "correlation({1 3 _ 6 6}, {6 6 4 2 3})"] all
::NAP::118-118  f32  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 1      Name: (NULL)    Coordinate-variable: ::NAP::119-119
Value:
-0.924138
 4.000000
```

Element 2 (base 0) of *x* is missing, so element 2 from *y* is not used and the sample size is 4 (as shown in the second element of the result). The correlation between {1 3 6 6} and {6 6 2 3} is calculated to be -0.924138.

If *x* and *y* have different shapes then the smaller of *x* and *y* is a window (*chip*) array which is moved around in the other array, producing a correlation for each position.
lag_0 is vector of row lags (default: all possible)
lag_1 is vector of column lags (default: all possible)

Grid Functions

There is currently just one grid function, `invert_grid`, but it has variants for one and two dimensions. The function defines a piecewise (bi-)linear mapping as the inverse of a given piecewise (bi-)linear mapping.

In the 1D case, the function is called by `invert_grid(`*y*`,` *ycv*`)`,
where *y* is the known mapping (and has a coordinate variable corresponding to *x*)
and *ycv* is the desired new *y* coordinate variable.
We have a piecewise-linear mapping from *x* to *y*, and we want a piecewise-linear mapping from *y* to *x*. The following example starts with a mapping from *x* to *y* defined by the two lines joining the three points (0, 0), (2, 1) and (5, 4). It produces the inverse mapping from *y* to *x* defined by the four lines joining the five points (0, 0), (1, 0.5), (2, 1.5), (2.5, 1.5) and (3, 2).

```
% nap "y = {0 1 4}"
::NAP::90-90
% $y set coo "{0 2 5}"
% [nap "coordinate_variable(y) /// y"]
```

```
0 2 5
0 1 4
% [nap "ycv = 0 .. 2 ... 1r2"]
0 0.5 1 1.5 2
% nap "x = invert_grid(y,ycv)"
::NAP::106-106
% $x all
::NAP::106-106  f32  MissingValue: NaN  References: 1  Unit: (NULL)
Link: ::NAP::107-107
Dimension 0   Size: 5      Name: (NULL)    Coordinate-variable: ::NAP::103-103
Value:
0 1 2 2.5 3
% [nap "coordinate_variable(x) /// x"]
0.0 0.5 1.0 1.5 2.0
0.0 1.0 2.0 2.5 3.0
```

In the 2D case, the function is called by `invert_grid(`*y*`,` *ycv*`,` *x*,*xcv*`)`,
where matrix *y* defines a mapping from *ij* space to *y*.
matrix *x* defines a mapping from *ij* space to *x*.
The result is a 3D array whose

- dimension 0 is of size 2, corresponding to the *i* and *j* mappings
- Dimension 1 has the specified coordinate-variable *ycv*
- Dimension 2 has the specified coordinate-variable *xcv*.

We can think of the result as two matrices defining mappings from *xy* space to *i* and *j* respectively.

Functions related to Special Data-types

Function	Result
`open_box(`*x*`)`	NAO pointed to by boxed NAO *x*
`pad(`*x*`)`	Normal NAO corresponding to ragged NAO *x*
`prune(`*x*`)`	Ragged NAO corresponding to normal NAO *x*

The following example illustrates the use of the function `open_box`, which allows one to extract NAOs from a structure created with the operator "`,`".

```
% nap "pointers = {4 5} , 'hello' , 9"
::NAP::9776-9776
% $pointers
9772 9773 9775
% [nap "open_box(pointers(0))"] all
::NAP::9772-9772  i32  MissingValue: -2147483648  References: 1  Unit: (NULL)
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Value:
4 5
```

```
% [nap "open_box(pointers(1))"] all
::NAP::9773-9773  c8  MissingValue: (NULL)  References: 1  Unit: (NULL)
Dimension 0   Size: 5      Name: (NULL)    Coordinate-variable: (NULL)
Value:
hello
% [nap "open_box(pointers(2))"] all
::NAP::9775-9775  i32  MissingValue: -2147483648  References: 1  Unit: (NULL)
Value:
9
```

The following example illustrates the use of functions `prune` and its inverse `pad`. Function `prune` creates a `ragged` array. This suppresses missing values at the start and end of the least significant dimension (column in this matrix case). In this matrix case it creates a separate NAO for each row and stores an index (slot number) to these in the result.

```
% nap "data = {{0 1.5 2 -1}{_ 1 4 1n}{4#_}{2#_ 9 -9}}"
::NAP::9736-9736
% $data
 0.0  1.5  2.0 -1.0
   _  1.0  4.0    _
   _    _    _    _
   _    _  9.0 -9.0
% nap "compressed_data = prune(data)"
::NAP::9738-9738
% $compressed_data all
::NAP::9738-9738  ragged  MissingValue: 0  References: 1  Unit: (NULL)
Dimension 0   Size: 4      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 4      Name: (NULL)    Coordinate-variable: (NULL)
Value:

0   start-index: 0   ::NAP::9740-9740
1   start-index: 1   ::NAP::9741-9741
2   start-index: 4   ::NAP::9742-9742
3   start-index: 2   ::NAP::9743-9743
% ::NAP::9743-9743
9 -9
% [nap "pad(compressed_data)"] all
::NAP::9745-9745  f64  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 4      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 4      Name: (NULL)    Coordinate-variable: (NULL)
Value:
 0.0  1.5  2.0 -1.0
   _  1.0  4.0    _
   _    _    _    _
   _    _  9.0 -9.0
```

Morphological Functions

Function	Result
`dilate(`*x*,*se*[,*seo*])	Binary dilation of *x*; *se* = structure-element; *seo* = origin of structure-element
`erode(`*x*,*se*[,*seo*])	Binary erosion of *x*; *se* = structure-element; *seo* = origin of structure-element
`moving_range(`*x*,*s*`)`	Range (max-min) of moving shape-*s* window around matrix *x*

Morphological Binary Dilation and Erosion

x is an *n* by *m* non-negative matrix that is being dilated or eroded.
se is the morphological structure element, an *a* by *b* matrix, where $a<n$ and $b<m$.
seo is the origin of the structure element indexed from 0 at the top left corner.

Moving Range

Move a window over the matrix *x* and find the maximum difference between values in the moving window. The result is placed in the element nearest the centre of the moving window.

Statistical Functions

These are defined in `stat.tcl`. Note that there is also a built-in statistical function called `correlation`, which is documented in [NAP Built-in Functions].

Three examples are provided for each function. The first uses the vector `v` which is defined as follows (note the missing values):

```
% nap "v = {12 6 _ 7 3 15 _ 10 18 5}"
::NAP::14-14
% $v all -c -1
::NAP::14-14  i32  MissingValue: -2147483648  References: 1  Unit: (NULL)
Dimension 0   Size: 10     Name: (NULL)    Coordinate-variable: (NULL)
Value:
12 6 _ 7 3 15 _ 10 18 5
```

The second example produces statistics of each column of a matrix. The second example produces statistics of each row of a matrix. The matrix is called `m` and is defined as follows:

```
% nap "m = {
    {1.5 2.1 -5}
    {5.5 9.4  0}
}"
::NAP::15-15
% $m all -c -1
::NAP::15-15  f64  MissingValue: NaN  References: 1  Unit: (NULL)
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
 1.5  2.1 -5.0
 5.5  9.4  0.0
```

am(*x*[,*verb_rank*])

Arithmetic mean.

```
% [nap "am(v)"]
9.5
% [nap "am(m)"]
3.5 5.75 -2.5
% [nap "am(m,1)"]
-0.466667 4.96667
```

median(*x*[,*verb_rank*])

Median.

```
% [nap "median(v)"]
8.5
% [nap "median(m)"]
3.5 5.75 -2.5
% [nap "median(m,1)"]
1.5 5.5
```

gm(*x*[,*verb_rank*])

Geometric mean.

```
% [nap "gm(v)"]
8.19852
% [nap "gm(m)"]
2.87228 4.44297 0
% [nap "gm(m,1)"]
_ 0
```

rms(*x*[,*verb_rank*])

Root mean square.

```
% [nap "rms(v)"]
10.6771
% [nap "rms(m)"]
4.03113 6.81065 3.53553
% [nap "rms(m,1)"]
3.24859 6.28782
```

sd(*x*[,*verb_rank*])

Standard-deviation (with division by n).

```
% [nap "sd(v)"]
4.8734
% [nap "sd(m)"]
2 3.65 2.5
% [nap "sd(m,1)"]
```

```
3.2149 3.85602
```

sd1(*x*[,*verb_rank*])

Standard-deviation (with division by n-1).

```
% [nap "sd1(v)"]
5.20988
% [nap "sd1(m)"]
2.82843 5.16188 3.53553
% [nap "sd1(m,1)"]
3.93743 4.72264
```

var(*x*[,*verb_rank*])

Variance (with division by n).

```
% [nap "var(v)"]
23.75
% [nap "var(m)"]
4 13.3225 6.25
% [nap "var(m,1)"]
10.3356 14.8689
```

var1(*x*[,*verb_rank*])

Variance (with division by n-1).

```
% [nap "var1(v)"]
27.1429
% [nap "var1(m)"]
8 26.645 12.5
% [nap "var1(m,1)"]
15.5033 22.3033
```

Miscellaneous Functions

These are defined in `nap_function_lib.tcl`.

acof2boxed(*filename*)

Read ascii cof (.acof) file and return boxed nao.

Example:
```
nap "i = acof2boxed('abc.acof')"
```

ap0(*from*,*to*,*step*)

Quasi-arithmetic-progression, whose final step may be short.

Example:
```
% [nap "ap0(0,8,3)"]
0 3 6 8
```

ap_n(*from*,*to*,*n*)

Arithmetic progression with *n* elements.

Example:
```
% [nap "ap_n(0,8,4)"]
0 2.66667 5.33333 8
```

color_wheel(*n*,*v*,*b*)

Square containing color wheel.
n is number rows & columns
v is desired "value" level
b is background colour outside circle

Example:
```
nap "color_wheel(100,255,3#150)"
```
This produces a `u8` array with shape `{3 100 100}` & values from 0 to 255.

cv(*main_nao*[,*dim_number*])

This is simply an alias for `coordinate_variable`.

fix_longitude(*longitude*)

Adjust elements of longitude vector by adding or subtracting multiple of 360 to ensure:
$-180 \leq x_0 < 180$
$0 \leq x_{i+1} - x_i < 360$
Also ensure unit is `degrees_east`.

fuzzy_floor(*x*[,*eps*])

Like `floor()` except allow for rounding error.
eps is tolerance and defaults to `1e-9`.

Example:
```
% [nap "fuzzy_floor({4.998 4.9998},1e-3)"]
4 5
```

fuzzy_ceil(*x*[,*eps*])

Like `ceil()` except allow for rounding error.
eps is tolerance and defaults to `1e-9`.

Example:
```
% [nap "fuzzy_ceil({5.002 5.0002},1e-3)"]
6 5
```

gets_matrix(*filename*)

Read text file and return NAO matrix whose rows correspond to the lines in the file. Ignore blank lines and lines whose first non-whitespace character is '#'.

Example:
```
gets_matrix('my_matrix.txt')
```

head(*x*[,*n*])

If $n \geq 0$ then result is 1st n elements of x, cycling if $n >$ nels(x).
n defaults to 1.

If *n* < 0 then result is 1st nels(*x*)+*n* elements of *x* i.e. drop –*n* from end

Example:

```
% [nap "head({3 1 9 2 7})"]
3
% [nap "head({3 1 9 2 7}, 2)"]
3 1
% [nap "head({3 1 9 2 7}, -2)"]
3 1 9
```

hsv2rgb(*hsv*)

Convert colour in HSV form to RGB.
hsv is an array whose leading dimension has size 3.

Layer 0 along this dimension corresponds to hue as an angle in degrees. Angles of any sign or magnitude are allowed. Red = 0, yellow = 60, green = 120, cyan = 180, blue = –120, magenta = –60.

Layer 1 along this dimension corresponds to saturation in range 0.0 to 1.0.

Layer 2 along this dimension corresponds to "value". This has the same range as the RGB values, normally either 0.0 to 1.0 or 0 to 255. If you are casting the result to an integer & want a maximum of 255 then set the maximum to say 255.999. Otherwise you will get few if any 255s.

The result has the same shape as the argument (hsv).

See Foley, vanDam, Feiner and Hughes, Computer Graphics *Principles and Practice*, Second Edition, 1990, ISBN 0201121107 page 593.

Example:

```
% [nap "hsv2rgb {180.0 0.5 100.0}"]
50 100 100
```

isMissing(*x*)

1 if *x* misssing, 0 if present.

Example:

```
% [nap "isMissing {0 _ 9}"]
0 1 0
```

isPresent(*x*)

0 if *x* misssing, 1 if present.

Example:
```
% [nap "isPresent {0 _ 9}"]
1 0 1
```

merid_wt(*longitude*)

Calculate normalised (so sum weights = 1) meridional weights from longitudes.

Example:
```
% [nap "merid_wt {-180 -90 -45 0 90 180}"]
0.125 0.1875 0.125 0.1875 0.25 0.125
```

mixed_base(*x*,*b*)

Convert scalar value *x* to mixed base defined by vector *b*.

Following example converts 87 inches to yards, feet & inches:
```
% [nap "mixed_base(87, {3 12})"]
2 1 3
```

outer(*dyad*,*y*[,*x*])

Tensor outer-product.

dyad is name of either

- function with two arguments
- binary (dyadic) operator

x is vector
y is vector defaulting to *x*
Result is cross-product of *x* and *y*, applying *dyad* to each combination of *x* & *y*.
x & *y* are the coordinate variables of the result.

Following example produces a multiplication table:

```
% [nap "outer('*', 1 .. 5)"]
 1  2  3  4  5
 2  4  6  8 10
 3  6  9 12 15
 4  8 12 16 20
 5 10 15 20 25
```

reverse(*x*[,*verb_rank*])

Reverse order of items in array *x*.

Examples:

```
% [nap "reverse {1 9 0 7}"]
7 0 9 1
% [nap "reverse {{1 2 3}{4 5 6}}"]
4 5 6
1 2 3
% [nap "reverse({{1 2 3}{4 5 6}}, 1)"]
3 2 1
6 5 4
```

scaleAxis(*xstart*,*xend*[,*nmax*[,*nice*]])

Find suitable values for axis of graph.
xstart: 1st data value
xend: Final data value
nmax: Max. allowable number of elements in result (Default: 10)
nice: Allowable increments (Default: `{1 2 5}`)
Result is the arithmetic progression which:

- is within interval from *xstart* to *xend*
- has same order (ascending/descending) as *xstart*`//`*xend*
- has increment equal to element of *nice* times a power `(-30 .. 30)` of 10
- has at least two elements
- has no more than *nmax* elements if possible
- has as many elements as possible. (Ties are resolved by choosing earlier element in *nice*.)

Example:

```
% [nap "axis = scaleAxis(-370, 580, 10, {10 20 25 50})"] value
-300 -200 -100 0 100 200 300 400 500
```

`scaleAxisSpan(`*xstart*`,`*xend*`[,`*nmax*`[,`*nice*`]])`

Find suitable values for axis of graph.
xstart: 1st data value
xend: Final data value
nmax: Max. allowable number of elements in result (Default: 10)
nice: Allowable increments (Default: `{1 2 5}`)
Result is the arithmetic progression which:

- includes the interval from *xstart* to *xend*
- has same order (ascending/descending) as *xstart*`//`*xend*
- has increment equal to element of *nice* times a power `(-30 .. 30)` of 10
- has at least two elements
- has no more than *nmax* elements if possible
- has as many elements as possible. (Ties are resolved by choosing earlier element in *nice*.)

Example:

```
% [nap "axis = scaleAxisSpan(-370, 580, 10, {10 20 25 50})"] value
-400 -200 0 200 400 600
```

`range(`*a*`)`

Result is 2-element vector containing minimum and maximum of array *a*.
Example:

```
% [nap "range {{9 -1 -5}{2 9 3}}"]
-5 9
```

`tail(`*x*`[,`*n*`])`

If *n* ≥ 0 then result is final *n* elements of *x*, cycling if *n* > nels(*x*).
n defaults to 1.
If *n* < 0 then result is final `nels(`*x*`)+`*n* elements of *x* i.e. drop –*n* from start.

Example:

```
% [nap "tail({3 1 9 2 7})"]
7
% [nap "tail({3 1 9 2 7}, 2)"]
2 7
% [nap "tail({3 1 9 2 7}, -2)"]
9 2 7
```

`zone_wt(`*latitude*`)`

Calculate normalised (so sum weights = 1) zonal weights from latitudes.

Example:

```
% [nap "zone_wt(-90 .. 90 ... 30)"] value
0.0334936 0.125 0.216506 0.25 0.216506 0.125 0.0334936
```

Binary Input/Output Procedures

These are oriented to Fortran unformatted IEEE files. They are defined in `bin_io.tcl`.

The `cif` (`conmap` input file) format is one which originated in the Melbourne University Department of Meteorology in the days before netCDF and HDF. It is now rather obsolete but is still used within CSIRO and other Australian organisations. A cif is a Fortran unformatted IEEE file consisting of one or more frames, each of which consists of six records as follows:

- number of rows
- vertical coordinate variable (often latitude)
- number of columns
- horizontal coordinate variable (often longitude)
- title
- main data (matrix)

Procedure `size_of` *dataType*

Number of bytes in *dataType*.

Example:

```
% size_of i16
2
```

Procedure `get_bin` *dataType* [*fileId*]

Read next Fortran binary (unformatted) record.
dataType: `c8`, `u8`, `u16`, `u32`, `i8`, `i16`, `i32`, `f32` or `f64`
fileId: Tcl file handle (default: `stdin`)

The example for `put_bin` also acts as an example for `get_bin`.

Procedure `put_bin` *nap_expr* [*fileId*]

Write Fortran binary (unformatted) record.
nap_expr: NAP expression to be evaluated in caller namespace
fileId: Tcl file handle (default: `stdout`)

The following example creates a NAO called `squares`, writes it to a file, then reads the data from this file into a NAO called `in`.

```
% nap "squares = (0 .. 4)**2"
::NAP::66-66
% $squares all
```

```
::NAP::66-66  f32  MissingValue: NaN  References: 1  Unit: (NULL)
Dimension 0   Size: 5      Name: (NULL)    Coordinate-variable: (NULL)
Value:
0 1 4 9 16
% set file [open tmp.bin w]; # open file "tmp.bin" for writing
file5
% put_bin squares $file; # write data from squares
% close $file
% set file [open tmp.bin]; # open file "tmp.bin" for reading
file5
% nap "in = [get_bin f32 $file]"; # read data into in
::NAP::78-78
% close $file
% $in all
::NAP::78-78  f32  MissingValue: NaN  References: 1  Unit: (NULL)
Dimension 0   Size: 5      Name: (NULL)    Coordinate-variable: (NULL)
Value:
0 1 4 9 16
```

Procedure `get_cif1` [*options*]

Read next frame from cif (Conmap Input File).
Options:
`-f` **fileId**: Tcl file handle (default: `stdin`)
`-g 0|1`: `1` (default) for geographic mode, `0` for non-geographic mode
`-m` **real**: Input missing value (default: `-7777777.0`)
`-um` **text**: Units for matrix (default: none)
`-ux` **text**: Units for x (default: if geographic mode then `degrees_east`, else none)
`-uy` **text**: Units for y (default: if geographic mode then `degrees_north`, else none)
`-x` **text**: Name of dimension x (default: if geographic mode then `longitude` else x)
`-y` **text**: Name of dimension y (default: if geographic mode then `latitude` else x)

The following example reads the first frame of a cif into a NAO called `in`, then displays it (including the coordinate variables).

```
% set f [open 7.cif]
file5
% nap "in = [get_cif1 -f $f]"
::NAP::218-218
% close $f
% $in all
::NAP::218-218  f32  MissingValue: NaN  References: 1  Unit: (NULL)
This data originated from ascii conmap input file 'acif.7'
Dimension 0   Size: 3      Name: latitude  Coordinate-variable: ::NAP::97-97
Dimension 1   Size: 4      Name: longitude  Coordinate-variable: ::NAP::169-169
Value:
 1  1  2 -3
 1  _  3 -4
```

```
 2  0  4  5
% ::NAP::97-97
-60 30 60
% ::NAP::169-169
-90 30 90 180
```

Procedure `get_cif` [*options*]

Read first frame from cif (Conmap Input File).
Options:
`-f` **filename**: file name (default: `""` which is treated as `stdin`)
`-g 0|1`: `1` (default) for geographic mode, `0` for non-geographic mode
`-m` **real**: Input missing value (default: `-7777777.0`)
`-um` **text**: Units for matrix (default: none)
`-ux` **text**: Units for x (default: if geographic mode then `degrees_east`, else none)
`-uy` **text**: Units for y (default: if geographic mode then `degrees_north`, else none)
`-x` **text**: Name of dimension x (default: if geographic mode then `longitude` else x)
`-y` **text**: Name of dimension y (default: if geographic mode then `latitude` else x)

The following example reads the first frame of a cif into a NAO called `in`, then displays it (including the coordinate variables).

```
% nap "in = [get_cif -f 7.cif]"
::NAP::357-357
% $in all
::NAP::357-357  f32  MissingValue: NaN  References: 1  Unit: (NULL)
This data originated from ascii conmap input file 'acif.7'
Dimension 0   Size: 3      Name: latitude  Coordinate-variable: ::NAP::236-236
Dimension 1   Size: 4      Name: longitude  Coordinate-variable: ::NAP::308-308
Value:
 1  1  2 -3
 1  _  3 -4
 2  0  4  5
% [nap "coordinate_variable(in,0)"]
-60 30 60
% [nap "coordinate_variable(in,1)"]
-90 30 90 180
```

Procedure `get_nao` [*fileName* [*dataType* [*shape*]]]

Read NAO from (binary) nao file.
filename: file name (default: `""` which is treated as `stdin`)
dataType: `c8`, `u8`, `u16`, `u32`, `i8`, `i16`, `i32`, `f32` or `f64`
shape: shape of result (Default: number of elements until end)

Procedure `put_cif1` [*nap_expr* [*fileId*]]

Write NAO as frame of cif.
nap_expr: NAP expression to be evaluated in caller namespace
fileId: Tcl file handle (default: `stdout`)

Procedure `put_cif` [*nap_expr* [*fileName*]]

Write NAO as entire cif.
nap_expr: NAP expression to be evaluated in caller namespace
fileName: file name (default: `stdout`)

Procedure `put_nao` [*nap_expr* [*fileName*]]

Write NAO to (binary) nao file.
nap_expr: NAP expression to be evaluated in caller namespace
fileName: file name (default: `stdout`)

Map Projection Procedures

These are defined in `projection.tcl`.

`projection` *code p0 p1 p2* ...

Define functions `projection_x` and `projection_y` for specified map projection.

code is map projection code (default: `CylindricalEquidistant`) as follows:

- `CylindricalEquidistant`: *p0* = x-origin (default: `""`)
- `Mercator`: *p0* = x-origin (default: `""`)
- `NorthPolarEquidistant`: North Polar azimuthal equidistant
- `SouthPolarEquidistant`: South Polar azimuthal equidistant
- `SouthPolarStereographic`: As used by IASOS

p0 p1 p2 … define parameters of the projection. Some projections use *p0* to specify an "x-origin". This is the minimum result to be returned by projection_x. If x-origin is `""` then there is no defined minimum result.

N-dimensional Array Objects (NAOs)

A NAO is a data structure in memory. A NAO consists of the following components:

slot number
: Index of entry in internal table used to provide fast access to NAOs.

sequence number
: Number (starting from 1) assigned in order of creation of NAOs.

OOC-name
: Name of object-oriented command associated with NAO. Also used as unique identifier of NAO.

reference count
: Number of Tcl variables, NAOs, windows, etc. pointing to this NAO. If the count decrements to 0 then NAP deletes the NAO and its associated OOC.

nap_cd
: pointer to NAP structure created for each interpreter.

dataType
: one of the following:

Name	Description
`c8`	8-bit character
`i8`	8-bit signed integer
`i16`	16-bit signed integer
`i32`	32-bit signed integer
`u8`	8-bit unsigned integer
`u16`	16-bit unsigned integer
`u32`	32-bit unsigned integer
`f32`	32-bit floating-point
`f32`	64-bit floating-point
`ragged`	compressed
`boxed`	slot numbers (used as pointers to NAOs)

step
: An efficiency hint for searching vectors. If this component is undefined before a search then NAP defines it according to whether the vector is
  - unordered
  - sorted into ascending order
  - sorted into descending order
  - quasi-arithmetic-progression i.e. a vector all of whose steps are equal, except possibly the final one which may be shorter.

mortal

This is set FALSE for NAOs (e.g. standard missing values) which must not be deleted regardless of reference count.

label
: Text containing title, description of data, etc.

unit
: Text defining unit of measure.

rank
: number of dimensions.

nels
: number of elements = product(shape).

nbytes
: number of bytes in NAO. Mainly for debugging.

link slot
: slot number (0 = none) of link NAO, which can be used to attach further information to a NAO (possibly via a boxed NAO which could link to any number of further NAOs).

next slot
: used internally to create NAO *death list* when executing a Tcl procedure defining a NAP function. (0 = none)

missing value slot
: slot number (0=none) of missing-value NAO.

pointer to missing value
: pointer to missing value NAO (for fast access).

pointer to missing value function
: This function tests whether an element of a NAO is missing.

ragged start slot number
: slot number (0=none) of vector NAO giving start index of each row of ragged array.

shape
: sizes of dimensions.

dimension names
: names (if any) of dimensions.

coordinate-variable slot numbers
: slot numbers (0 = none) of CVs.

data
: Main data.

Interfacing NAP to a DLL based on C or Fortran Code

The file `make_dll.tcl` defines procedures for automatically producing an interface from NAP to a *DLL* (*dynamic-link library* or *shared library*) based on C or Fortran Code. This process defines a new tcl command which can either be used directly or via another interface (written in Tcl) defining a NAP function.

`make_dll` *options newCommand argDec argDec argDec* ...

This is the standard procedure used to create a DLL.

newCommand is name of new command.

Each argument-declaration *argDec* is a list with the form `{`*name dataType intent*`}` where

- *name* is any string (used only in error messages)
- *dataType* can be: `c8 i8 u8 i16 u16 i32 u32 f32 f64 void`
- *intent* can be: `in` (default) or `inout`. Actual `in` arguments can be expressions of any type (including `ragged`) and will be converted to the specified type (unless this is `void`).

options are:
`-quiet`: Do not echo commands.
`-compile` **command**: C compile-command with options
`-dll` **fileName**: output filename for DLL (default: *newCommand*.dll for windows, *newCommand*.so for unix)
`-entry` : User-routine entry-point (default: *newCommand*). Note that fortran entry points often include suffix '_'.
`-header` **fileName**: header (`*.h`) filename (default: none)
`-libs` **fileNames**: filenames of extra binary libraries (default: none)
`-link` **command**: Link-command with options
`-object` **fileName**: User-routine object-file (default: *newCommand*`.obj` for windows, *newCommand*`.o` for unix)
`-source` **fileName**: Output file containing C source code of interface (default: *newCommand*`_i.c`)
`-version` : Version number (default: `1.0`)

The following example (under Linux) defines a new NAP function `partialProd` which calculates partial-products. This is analogous to the standard nap function `psum` which calculates partial sums. The new function is based on the following C file `pprod.c`:

```
void pprod(int *n, float *x, float *result) {
   int         i;
   float       prod = 1;

   for (i = 0; i < *n; i++) {
       result[i] = prod = prod * x[i];
   }
}

% exec cc -c -o pprod.o pprod.c
% make_dll pprod {n i32 in} {x f32} {y f32 inout}
cc -I/home/dav480/tcl/include -c pprod_i.c
ld -shared -o libpprod.so pprod_i.o pprod.o
% load ./libpprod.so
% proc partialProd x {
    set result [nap "reshape(0f, shape(x))"]
    pprod  "nels(x)" x result
    return $result
}
% [nap "partialProd({2 1.5 3 0.5})"]
2 3 9 4.5
```

You can do the same thing in fortran 90 using the following source code:

```
subroutine pprod(n, x, result)
   integer, intent(in) :: n
   real, intent(in) :: x(n)
   real, intent(out) :: result(n)
   integer :: i
   real :: prod
   prod = 1.0
   do i = 1, n
       prod = prod * x(i)
       result(i) = prod
   end do
end subroutine pprod
```

The following log was produced using the Linux Lahey f95 compiler. (Note that the entry point is `pprod_`)

```
% exec lf95 -c pprod.f90
Compiling file pprod.f90.
Compiling program unit pprod at line 1:
```

```
/home/dav480/tmp/asm03529aaa.s: Assembler messages:
/home/dav480/tmp/asm03529aaa.s:86: Warning: translating to `fstp %st'
/home/dav480/tmp/asm03529aaa.s:90: Warning: translating to `fstp %st'
% make_dll -entry pprod_ pprod {n i32 in} {x f32} {y f32 inout}
cc -I/home/dav480/tcl/include -c pprod_i.c
ld -shared -o libpprod.so pprod_i.o pprod.o
% load ./libpprod.so
% proc partialProd x {
    set result [nap "reshape(0f, shape(x))"]
    pprod  "nels(x)" x result
    return $result
}
% [nap "partialProd({2 1.5 3 0.5})"]
2 3 9 4.5
```

`make_dll_i` *options newCommand argDec argDec argDec* ...

Make NAP C interface to user's C function or Fortran subroutine. This procedure is normally used via `make_dll`, but may be used directly if you prefer to do your own compiling and linking. The result of `make_dll_i` is the C code.

The arguments are similar to `make_dll`, except that the only options are:
`-entry` : User-routine entry-point (default: *newCommand*). Note that fortran entry points often include suffix '_'.
`-header` **fileName**: header (`*.h`) filename (default: none)
`-version` : Version number (default: `1.0`)

NAP Photo Image Format

NAP defines a new photo image format. This enables one to use the "`image create photo`" command to produce a photo image from a NAO. One can also use the photo image write operation to produce a NAO from a photo image.

The data type of the NAO is normally `u8` (8-bit unsigned integer). Any other type will be converted to `u8`.

The rank can be 2 or 3. A matrix gives a grey-scale image. Colour requires three dimensions. In this case there normally are three layers corresponding to red, green and blue. If there are only two layers then the first is used for both red and green (which together give yellow).

The name of the new photo image format is "`NAO`".

The following example (input only shown) creates and displays a grey-scale photo image from a `u8` matrix:

```
nap "u = u8(reshape(0 .. 255, 2 # 255))"
set i [image create photo -format NAO -data $u]
button .b -image $i
pack .b
```

The following example (input only shown) creates and displays a colour photo image from a three-dimensional `u8` array. It then writes this image to a GIF file named "`n.gif`".

```
destroy .b
nap "u = u8(reshape({32768#0 65536#255}, {3 2#256}))"
set i [image create photo -format NAO -data $u]
button .b -image $i
pack .b
$i write n.gif -format GIF
```

The following example (input and output shown) first creates a photo image by reading this GIF file named "`n.gif`". Then a new NAO is created and assigned the name "`abc`".

```
% set pi [image create photo -file n.gif]
image8
% $pi write abc -format NAO
% $abc header
::NAP::2790-2790  u8  MissingValue: (NULL)  References: 1  Unit: (NULL)
```

```
Dimension 0   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 256    Name: (NULL)    Coordinate-variable: (NULL)
Dimension 2   Size: 256    Name: (NULL)    Coordinate-variable: (NULL)
```