Nap 6.2 User's Guide

Harvey Davies
CSIRO Marine and Atmospheric Research
Melbourne, Australia

October 2, 2006

Contents

Chapter 1

Introduction

1.1 Overview of Nap (N-Dimensional Array Processor)

Nap is a loadable extension of Tcl. Nap 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.

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 C format
- 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:

- 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)
- array facilities (constants, operators, functions, indexing)

Array indices can take fractional values, which give results defined by n-dimensional linear interpolation. Index values can be specified indirectly via coordinate-variable values (e.g. latitudes and longitudes).

The `nap_get` command creates a NAO from data read from a binary, HDF or netCDF file. Some platforms (only Linux on Intel 386 at the time of writing) support reading of remote virtual netCDF files provided by OPeNDAP (a.k.a. *DODS*) web servers.

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

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.

The *Caps/Nap GUI* provides browsers for:

- Tcl variables
- image files (e.g. GIF, JPEG)
- AVHRR satellite files (Caps package required)
- ATSR satellite files (Caps package required)
- CIF files (Melbourne University format)
- HDF/netCDF files

The HDF/netCDF browser is a convenient tool for quickly browsing HDF and netCDF files. Selected data can be

- displayed as text
- graphed
- shown as various kinds of images and maps
- animated
- used to create a NAO for further processing using NAP

1.2 Typographic Conventions

Hyperlinks are blue, as in tcl-nap. Internal references are red, as in section 1.1. Both can be clicked on if you are reading this on a screen rather than paper. Try it.

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`'.

1.3 Acknowledgments

Ken Iverson (who died in 2004) is the father of array processing. Nap does not use the radical mathematical conventions of his languages APL, and J, but he must be acknowledged as the source of the fundamental concepts upon which NAP is based. Nap has also adopted various J conventions such as that for floating-point constants (e.g. allowing 0.5 to be written in rational form as `1r2` and allowing powers of π as in `1p1`).

Nap would never have happened if the author (Harvey Davies) had not worked with Rhys Francis and Ian Mathieson on development of the data-parallel modelling language *DPML* and learned from them things like *yacc*. This project met a premature death but we did learn a lot. It is hoped to implement unit calculus (automatic unit conversion and definition) in Nap as was done in DPML.

The other strong influence on Nap was netCDF. It is amazing that a portable array file format did not exist until Russ Rew and the late Glenn Davis developed netCDF in the early 1990s. This work was based on CDF, which was developed by Michael Gough and Lloyd Treinish. Further details of the history of CDF and netCDF are available at
http://www.unidata.ucar.edu/software/netcdf/credits.html. Nap is designed to read and write netCDF (and the similar HDF) files and stores data in memory in stuctures called *NAOs*, which have similar properties to netCDF variables.

My CSIRO colleague Peter Turner made significant contributions to the Nap code. He wrote the C code for:

- NAO photo-image handler
- *draw* and *fill* OOC methods
- morphological functions (moving_range, dilate, erode)

He also made significant contributions to the following Tcl library files:

- `browse_var.tcl`
- `caps_nap_menu.html`
- `colour.tcl`
- `hdf.tcl`
- `pal.tcl`
- `plot_nao.tcl`
- `proc_lib.tcl`

The `nap_land_flag` command (see section 8.2) is based on code which was originally written by Dr. Chris. Mutlow at the Rutherford Appleton Laboratory in England. This code has been adapted for Nap by Peter Turner and Harvey Davies.

Past and present members of The Caps development group (Ian Grant, Edward King, Jenny Lovell, Paul Tildesley, Peter Turner and Chris Rathbone) have provided ideas, support and feedback over the years. Mark Collier and Janice Bathols were early users of Nap for processing results from atmospheric modelling. Their success and enthusiasm has has done a great deal to convert others to *napism*.

Dr Takeshi Enomoto, Earth Simulator Center, maintains the Nap Fink site, which provides distributions for the *Mac OS X* and *Darwin* platforms.

Chapter 2

Background

2.1 Demonstration of Simple Tcl/Tk

2.1.1 Introduction

The following are logs of demonstrations of simple *Tcl/Tk* (excluding Nap). These can be used as a starting point from which to explore the system.

2.1.2 First Look at Tcl

The following demonstrates Tcl syntax, variables and the following commands:

- `expr` (arithmetic)
- `set` (setting and displaying variables)
- `puts` (output)
- `for` (looping)

```
% expr 3.1416 * 10 * 10
314.16
% set pi 3.1416
3.1416
% set pi
3.1416
% set r 10
10
% expr $pi * $r * $r
314.16
% set area [expr $pi * $r * $r]
314.16
% puts "circle of radius $r has area $area"
circle of radius 10 has area 314.16
% puts "circle of radius $r has area [expr $pi * $r * $r]"
circle of radius 10 has area 314.16
% for {set r 4} {$r < 5.1} {set r [expr $r + 0.2]} {
puts "circle of radius $r has area [expr $pi * $r * $r]"
}
circle of radius 4 has area 50.2656
circle of radius 4.2 has area 55.417824
circle of radius 4.4 has area 60.821376
circle of radius 4.6 has area 66.476256
circle of radius 4.8 has area 72.382464
circle of radius 5.0 has area 78.54
```

2.1.3 First Look at Tk

Tcl was designed to be extended. One of the first extensions was *Tk*, which provides a toolkit for building GUIs (Graphical User Interfaces).

The following demonstration is intended to provide just a hint of the nature of Tk. It displays two buttons which can be pressed to execute simple commands.

```
% button .hello -text "press me" -command {puts "Hello world"}
.hello
% pack .hello
Hello world
% button .beep -text beep -command bell
.beep
% pack .beep
```

2.1.4 Syntax of Tcl

The syntax of Tcl is based on the following *metacharacters* (characters treated in special ways):
`; # $ " \ {} []`
Note that the following are *not* metacharacters:
`’ * ()`

```
% set s1 the; set s2 "This is $s1"; set s3 {end of it all.}
end of it all.
% puts "$s2$s3"
This is theend of it all.
% puts "${s1}2"; # Braces around name are needed when next char could be part of name
the2
% puts \
"$s2 $s3"
This is the end of it all.
% puts "$s2
$s3"
This is the
end of it all.
% puts "$s2\
$s3"
This is the end of it all.
% puts \
"$s2\n$s3"
This is the
end of it all.
% # This is a comment
% set s1; # Value of s1
the
% # Character # is only special at start:
% puts #
#
% puts [expr [string length $s2] + 1 + [string length $s3]]
26
% unset s1
% set s1
can’t read "s1": no such variable
% puts [expr "2 + 2"]
4
```

2.1.5 Lists and Strings

Tcl is a *text-oriented* language in which text can be treated as either a single *string* or a *list* of elements. A sentence is a simple *list* of words separated by white-space. An element of a list can itself be a list.

```
% set names "Bill Jan Harry"; # Simple list of 3 words
Bill Jan Harry
% lindex $names 0
Bill
% lindex $names 2
Harry
% lsort $names
Bill Harry Jan
% foreach name $names {puts "'$name' has [string length $name] letters"}
'Bill' has 4 letters
'Jan' has 3 letters
'Harry' has 5 letters
% set names {{Bill Jones} {Jan Smith} {Harry Brown}}; # list of lists
{Bill Jones} {Jan Smith} {Harry Brown}
% foreach name $names {puts "'$name' has [string length $name] letters"}
'Bill Jones' has 10 letters
'Jan Smith' has 9 letters
'Harry Brown' has 11 letters
% string toupper $names
{BILL JONES} {JAN SMITH} {HARRY BROWN}
% string map {i a rr r} $names; # change "i" to "a", "rr" to "r"
{Ball Jones} {Jan Smath} {Hary Brown}
% string index hello 0
h
% string index hello 4
o
% foreach name $names {puts "[lindex $name 1], [lindex $name 0]"}
Jones, Bill
Smith, Jan
Brown, Harry
% foreach name $names {
    foreach word $name {
       puts -nonewline [string index $word 0]
    }
    puts ""
}
BJ
JS
HB
```

2.1.6 File Input and Output

The following commands are demonstrated in the following log:

- `open`
- `close`
- `puts` (Write to file or screen)
- `gets` (Read from file)
- `format` (Controlled conversion of values to text)
- `glob` (List of files matching pattern)

```
% # Write file containing height and width on each line
```

```
% set lines {{2 4} {1 1} {9 3}}
{2 4} {1 1} {9 3}
% set f [open hw.txt w]; # open file for writing
filee1c290
% foreach line $lines {puts $f $line}
% close $f
%
% # Read this file and display: height, width, area of rectangle
% set f [open hw.txt]; # open file for reading
filee1cd80
% while {[gets $f line] >= 0} {
     set h [lindex $line 0]
     set w [lindex $line 1]
     puts "$h $w [expr $h * $w]"
}
2 4 8
1 1 1
9 3 27
% close $f
%
% format {%s is %5.2f} {my age} 60.127
my age is 60.13
% format {%s is %8.1f} {my age} 60.127
my age is     60.1
%
% # For files *.txt: Display lines containing "if"
% foreach file [glob *.txt] {
     set f [open $file]
     while {[gets $f line] >= 0} {
         if {[string match {*if*} $line]} {puts "$file $line"}
     }
}
```

2.1.7 Procedures and Control Structures

The following shows how the `proc` command can be used to define a procedure and thus a new command. It also illustrates the control-structure commands `if`, `for` and `foreach`.

```
% proc initials words {
    foreach word $words {append result [string index $word 0]}
    return $result
}
% initials {Bill Harry Jan}
BHJ
%
% # arithmetic progression
% proc ap {
     from
     to
     {step 1}
} {
     for {set next $from} {$next <= $to} {set next [expr $next + $step]} {
         lappend result $next
     }
     return $result
}
```

```
% ap 1 9 2
1 3 5 7 9
% ap 1 9
1 2 3 4 5 6 7 8 9
% ap 0 3 0.5
0 0.5 1.0 1.5 2.0 2.5 3.0
% foreach r [ap 1 9 2] {puts "square of $r is [expr $r * $r]"}
square of 1 is 1
square of 3 is 9
square of 5 is 25
square of 7 is 49
square of 9 is 81
%
% # Define factorial using recursion
% proc factorial n {
     if {$n > 1} {
         return [expr $n * [factorial [expr $n - 1]]]
     } else {
         return 1
     }
}
% factorial 3
6
% foreach i [ap 0 5] {puts "$i [factorial $i]"}
0 1
1 1
2 2
3 6
4 24
5 120
```

2.2 Data Models

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

2.2.1 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?

2.2.2 What is the measurement level?

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

Level	**Description**	**Valid Operations**	**Measure of Central Tendency**	**Examples**
nominal	Values denote categories which have no order	$= \neq$	mode	zip post-code chemical e.g. CO_2
ordinal	Values ordered Differences meaningless	$= \neq < \leq > \geq$	median	Richter earthquake scale
interval	Differences valid Quotients meaningless	$=\neq<\leq>\geq$ $+-$	arithmetic mean	temperature in °C
ratio	Quotients valid	$=\neq<\leq>\geq$ $+ - \times \div$	geometric mean	temperature in °K

2.2.3 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.

2.2.4 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*.

Some variables have a vector value at each point in space and time. For example, wind data is typically stored as two (east and north) components. These components are often treated as separate variables. An alternative approach is to have a single array whose dimensions include a *nominal* dimension corresponding to the components. Thus a wind array could include a dimension of size 2 corresponding to east and north.

Data located in a continuous space can be either *gridded* or *scattered*. Both types are discussed in section 2.4 (Nap Grids).

2.3 Terminology

2.3.1 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.

column final (least-significant) dimension

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.

2.3.2 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 $-\infty$) 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.

2.4 Grids

2.4.1 Dimensions, Coordinate variables and Mappings

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 (CV)*. This is a vector which defines a piecewise-linear mapping from subscript to physical dimension.

Nominal dimensions often have text labels associated with each value. For example, wind components could be labelled '`east`' and '`north`'. One could imagine some kind of CV containing such string values. Nap does allow a string CV, but there can be only a single character for each value. So in the wind case one would have to make do with a string CV with a value such as '`EN`'.

CVs are convenient when each array dimension maps to a single physical dimension. However, the relationship between array dimensions and physical dimensions may be more complex. Consider the example of a satellite image with dimensions *line* (row) and *pixel* (column). The physical dimension *latitude* depends on both array dimensions (*line* and *pixel*). There is a matrix (with the same *line* and *pixel* dimensions as the image) giving the latitude at each point. There is another similar longitude matrix. These two matrices define piecewise-bilinear mappings from line/pixel space to latitude/longitude space. The task of warping the image to latitude/longitude space is essentially that of defining mappings from latitude/longitude space to line/pixel space i.e. the inverses of the given mappings. This can be done using the Nap functions `invert_grid` and `invert_grid_no_trim`, which are discussed in section 5.5.10.

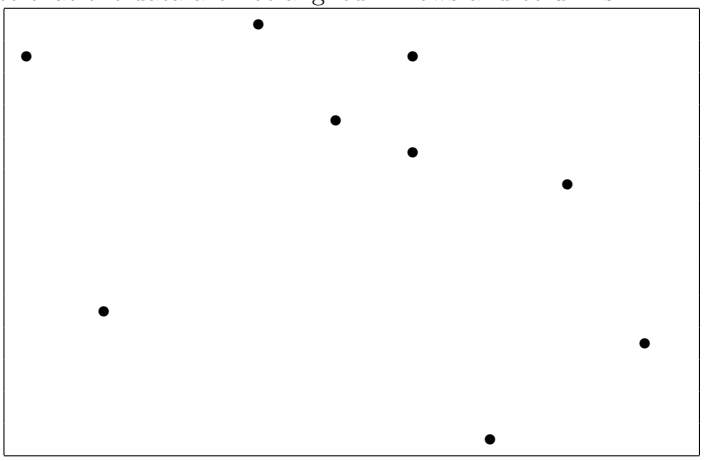
2.4.2 Difference between Grids and Scattered Data

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.

2.4.2.1 Example of Scattered 2D Data (Not a Grid)

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



2.4.2.2 Example of 2D Grid

The following example has the grid in black. The green 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		•	•		•	•	•

2.4.3 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.

2.4.4 Processing Scattered Data

Function `scattered2grid` interpolates scattered data onto a grid. See section 10.5.20 for details.

2.5 HDF and netCDF File Formats

HDF and netCDF are similar array-oriented file formats. Such files are popular in earth sciences such as meteorology and oceanography. They contain data referenced by symbol tables containing the names, data-types and dimensions of entities called *variables* in netCDF and *scientific data sets (SDSs)* in HDF. Each variable (SDS) can also have attributes such as a label, a format, a unit of measure and a missing-value.

The design of Nap has been strongly influenced by HDF and netCDF. Nap stores data in memory structures called *n-dimensional array objects* (*NAOs*) that have similar attributes to those of HDF SDSs and netCDF variables (e.g. label, format, unit of measure, missing-value). Nap has powerful high-level facilities for HDF and netCDF I/O.

Nap does not currently support the new HDF5 file format. Note that netCDF-4 software supports this HDF5 file format (as well as the traditional netCDF format) and is due for official release in late 2006. It is planned to support HDF5 and netCDF-4 in a future version of Nap.

2.6 References

2.6.1 Introduction

The following provides a guide to books, papers and web sites relevant to Nap. The opinions are those of the author (Harvey Davies).

2.6.2 Tcl/Tk

The following are my three favourite Tcl/Tk web sites:

wiki.tcl.tk Wiki-based open community site. Excellent starting point.

www.tcl.tk Main Tcl Developer Xchange site.

www.activestate.com/Products/ActiveTcl/ ActiveState provide ActiveTcl here for downloading.

Despite its age, I consider John Ousterhout's 1994 *Tcl and the Tk Toolkit* to still be the best introduction to Tcl/Tk. In fact it really is a classic example of clear readable technical writing. Ousterhout was the original developer of Tcl. Tcl has developed a lot since 1994 and the new features missing from this book are discussed in the Wiki.

The only detailed up-to-date book is the much less readable 4th edition of *Practical Programming in Tcl and Tk* by Brent Welch and Ken Jones with Jeffrey Hobbs. This book has the web site www.beedub.com/book.

2.6.3 Nap

The home page for *nap* (a.k.a. *tcl-nap*) is at http://tcl-nap.sourceforge.net/index.php. The main documentation of Nap is provided by the Nap User's Guide, which is what you are reading.

The only published paper on Nap is Harvey Davies's talk *The NAP (N-Dimensional Array Processor) Extension to Tcl* , which was given at the 9th (2002) Annual Tcl/Tk Conference. Both the original and a revised version are available.

Nap was originally developed as part of the CSIRO Caps project, which has the web page http://www.eoc.csiro.au/cats/caps/.

2.6.4 Array Processing

There are two main array processing professional associations in the world. The USA has the ACM *Special Interest Group on the APL and J languages*, which has the web page www.acm.org/sigapl. The British Computer Society has a specialist group the *British APL Association*, which publishes the journal *Vector*, which has the web page www.vector.org.uk. Both associations were originally concerned with only APL but now include other array processing languages.

The primary site for the J language is www.jsoftware.com.

2.6.5 Array-oriented File Formats

The primary site for netCDF is www.unidata.ucar.edu/packages/netcdf/index.html.
The primary site for HDF is hdf.ncsa.uiuc.edu.

2.6.6 Arithmetic

The classic article on floating-point arithmetic (and the IEEE Standard 754 for it) is David Goldberg's *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys, 1991. A more recent paper is Kahan's *Lecture Notes on the Status of IEEE 754* .

Chapter 3

Installation

3.1 Installing Tcl/Tk and Nap

3.1.1 Introduction

Tcl/Tk and *Nap* are available without charge in both binary (compiled) and source-code form. The following instructions explain how to install Tcl/Tk and then Nap. The Nap installer for Linux on the (64-bit) Intel IA64 installs the total system since this platform is not supported by ActiveState.

3.1.2 Installing Tcl/Tk

ActiveState Corporation provides free binary distributions of *Tcl/Tk* with many extensions. There are versions for Windows, Linux (on Intel 386), SunOS (Solaris) and HP-UX. HTML documentation is also available. These can all be downloaded from ActiveState Tcl Downloads.

Read the installation instructions. Note that these explain how to uninstall any existing version of Tcl/Tk, which should be done before installing the new one. (If this method of uninstalling fails then it is usually best to delete the whole *tcl root directory*.)

The *tcl root directory* is the directory in which Tcl is installed. Tcl extensions (e.g. Nap) are usually also installed in it. Note that it is denoted below by the variable *tcl_root*. A typical value for Windows is `C:\Program Files\Tcl`. Typical values for Unix (including Linux) are `~/tcl` (personal installation) and `/usr/local/tcl` (system installation).

Unix users should include *tcl_root*`/bin` in the list defined by the standard environment variable `PATH`. If there are problems with dynamic loading then try including *tcl_root*`/lib` in the list defined by the standard environment variable `LD_LIBRARY_PATH`.

The following three executables will be installed in *tcl_root*`/bin`:

`tclsh` This provides a bare-bones command-line *Tcl shell* without Tk. (Tk provides GUI facilities.) It is useful for non-interactive applications such as
- Unix shell scripts using Tcl
- automatically run (e.g. `cron`) jobs using Tcl

`wish` This *window shell* provides a full Tcl shell including Tk.

`tkcon` This provides all the facilities of `wish`, but in a better console. It is recommended as the normal way of running Tcl.

Each executable will execute a startup script file with the name listed in the following table, if this file is found in the home directory:

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

If you are unsure of which directory is *home*, then enter the following two commands:
`cd`
`pwd`

3.1.3 Tcl/Tk Documentation

The Caps/Nap Menu *help* menu includes the remote web Tcl Documentation.

The above installation process for Windows installs this documentation in the form of the compiled HTML help file `ActiveTclHelp.chm`. Under Windows, the above help menu does include this local copy of the documentation.

The standard installation for Unix installs `man` pages in the directory *tcl_root*`/man`. If you want to use these then include this directory in `MANPATH`.

It is also possible to create a local copy of this documentation in HTML form. Under Unix, this allows the Caps/Nap Menu (see section 12.2) to provide a local copy of the Tcl Documentation (similar to that for Windows). To do this, download the the HTML documentation into the directory *tcl_root*`/doc`. Then unpack the tar file and create a symbolic link *tcl_root*`/doc/html` pointing to the root directory of the unpacked files.

For example, if the Tcl root directory is `~/tcl` then download the version 8.4.9 file `ActiveTcl8.4.9.0.121397-html.tar.gz` into the directory `~/tcl/doc/`.
Change to this directory and then unpack the file using a command such as
`gtar zxf ActiveTcl8.4.9.0.121397-html.tar.gz`
or
`gunzip < gtar zxf ActiveTcl8.4.9.0.121397-html.tar.gz | tar xf -`
Then create the symbolic link using
`ln -s ActiveTcl8.4.9.0.121397-html html`

3.1.4 Installing Nap

Nap installers are available for:

- Linux on Intel 386
- Linux on IA64
- SunOS
- Windows

The *Mac OS X* and *Darwin* platforms are supported via the Fink project, which has a nap page.

It is hoped to support HP-UX in the near future. The SGI Irix platform is no longer supported.

Note that the Linux IA64 installer installs a total tcl system as well as Nap. This platform does not have an uninstaller, so it is suggested you install into a completely new directory rather than over an existing installation. For example, if your installation directory is `~/tcl` you could do the following:

```
mv ~/tcl ~/tcl_old
mkdir ~/tcl
```

The above installers can be downloaded from the page selected by clicking 'Files for Downloading' on http://tcl-nap.sourceforge.net/. They are in the form of *starpacks*, which are self-contained binary executables which contain the files to be installed. Starpacks are themselves based on Tcl and are described in *Beyond TclKit - Starkits, Starpacks and other *stuff* .

Then you simply execute the installer. It will first prompt you for the directory in which to install Nap, which is usually the *Tcl root directory*. Then it will check for existing Nap files and offer to delete them (strongly recommended) if any are found.

Finally it will offer to install three *standard Nap startup script files* (with the names in the above table) in your home directory. It is recommended that you install these unless you have alternative startup files. Such alternative files can include the following commands to facilitate use of Nap:
`package require nap`

```
namespace import ::NAP::*
```
If there will be other users then copy the three startup files to their home directories.

The script files `~/my_tkcon.cfg` and `~/my.tcl` will be sourced if they exist. You can create `~/my_tkcon.cfg` to tailor the tkcon console to your personal requirements. You can create `~/my.tcl` to tailor other aspects of the Tcl system to your personal requirements. See section 3.2 for examples.

These scripts and the standard Nap startup scripts are interrelated. The script `tkcon.cfg` (`.tkconrc`) sources `~/my_tkcon.cfg` and initialises tkcon to source `wishrc.tcl` (`.wishrc`). The script `wishrc.tcl` (`.wishrc`) in turn sources `tclshrc.tcl` (`.tclshrc`), which sources `~/my.tcl`.

On Unix systems it is possible to install Nap in a directory (which we will denote by *nap_root*) other than the *Tcl root directory*. If *nap_root* $\neq$ *tcl_root* then *nap_root*`/lib` must be included in the lists defined by:

- the standard Tcl variable `auto_path`
- the standard environment variable `LD_LIBRARY_PATH`.

For example, assume Nap is installed in `~/alt`. The following two commands are included in `tclshrc.tcl` or `.tclshrc`:

```
lappend auto_path ~/alt/lib
set env(LD_LIBRARY_PATH) ":~/alt/lib:$env(LD_LIBRARY_PATH)"
```

3.1.5 Nap User's Guide

The above installation process installs a copy of the Nap User's Guide as file *nap_root*`/lib/nap?.?/nap_users_guide.pdf` (where `?.?` is the version number). The Caps/Nap Menu (see section 12.2) *help* menu includes this local copy as well as the remote web copy at http://tcl-nap.sourceforge.net/nap_users_guide.pdf.

3.1.6 Nap Source Code

Source code for Nap is maintained using CVS and can be accessed by:

- Browsing CVS Source Repository
- Using a CVS client to access CVS Source Repository
- Downloading tar file from Tcl-Nap Files.

3.1.7 Related Packages

Nap now provides the functionality previously provided by the (now obsolete) separate packages *convert_date*, *inform* and *land_flag*. However Caps is still separate.

The Windows Nap distribution now includes the *ezprint* utility, which provides access to printers. No such separate printing package is needed for Unix. Previous versions of Nap used the *printer* package to provide access to printers on all platforms.

Previous versions of Nap provided an interface to the vector facility in the package BLT. This allowed use of the BLT *graph* command, which was used at that time by the Nap procedure *plot_nao*. Nap no longer provides an interface to BLT or uses it in any way.

3.2 Sample Startup Scripts

3.2.1 Introduction

These startup scripts tailor *Tcl* and *tkcon* to personal requirements. The scripts can be copied to your home directory and modified as desired.

3.2.2 Script `my.tcl`

The script `~/my.tcl` is called by the script `~/.tclshrc` or `~/tclshrc.tcl` that is created by the standard nap installation. This `~/my.tcl` script can be used to tailor Tcl in various ways such as

- initialising packages
- initialising libraries
- defining procedures
- sourcing other scripts

The following sample script initialises a personal library and defines some simple aliases:

```
# my.tcl --
#
# Initialise my library of Tcl procedures (including Nap functions) by
sourcing its tclIndex.
# The library consists of files in directory ~/my_tcl_lib
# The file tclIndex is created by the following command:
#     auto_mkindex ~/my_tcl_lib *.tcl
foreach dir {~/my_tcl_lib} {source $dir/tclIndex}
#
proc define_alias {alias args} {proc $alias args "eval $args \$args"}
#
define_alias e expr
define_alias n nap
define_alias p pwd
define_alias cp file copy
define_alias md file mkdir
define_alias rm file delete
```

3.2.3 Script `my_tkcon.tcl`

The script `~/my_tkcon.tcl` is called by the script `~/.tkconrc` or `~/tkcon.cfg` that is created by the standard nap installation. This `~/my_tkcon.tcl` script can be used to tailor various aspects of tkcon.

The following sample script defines

- the font used by tkcon
- the number of rows and columns provided by tkcon
- the number of commands in history
- the prompt at start of each input line

```
# my_tkcon.cfg --
#
set ::tkcon::OPT(font) "Courier 10"
set ::tkcon::OPT(rows) 40
set ::tkcon::OPT(cols) 80
set ::tkcon::OPT(history) 200; # number of commands in history
set ::tkcon::OPT(prompt1) {% };# prompt at start of each input line
# set ::tkcon::OPT(prompt1) {[history nextid] % };# alternative
```

Chapter 4

Nap Basics

4.1 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`' displays 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
- 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 2.5 to 7. Note that element 1 is the second element because the subscript origin is 0 (as in other aspects of Tcl) rather than 1 (as in languages such as Fortran).

```
% $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*. The six commands do the following:

1. Assign to the variable `score` a 32-bit floating-point vector containing the five values 56, 75, 47, 99 and 49.
2. Display `score`.
3. Index a vector by a scalar '`2`' to give a scalar.
4. Index a vector by a vector '`{2 0 4}`' to give a vector.
5. Illustrate the operator '`..`' which defines an *arithmetic progression*.
6. Use such an arithmetic progression as an index.

```
% 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
% [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 '$\sum$'
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`' (meaning *false*) or '`1`' (meaning *true*).

4.2 NAOs, OOCs and nap command

The standard Tcl `expr` command produces a single (scalar) number. For example:

```
% expr "2 * 3.5"
7.0
```

Note that the Tcl result of this command is simply the text '`7.0`'.

However the `nap` command often produces an array containing millions of numbers. It is not practical to store, process and display millions of numbers as text. It is far better to store and process them in binary form. The binary values are stored in a memory object called an *N-dimensional Array Object (NAO)*, which also includes other information such as the number of elements and the data-type. Nap is designed to efficiently handle large arrays, but let us begin demonstrating it on the above simple scalar expression:

```
% nap "2 * 3.5"
::NAP::16-16
```

What is this strange Tcl result '`::NAP::16-16`'? It is the ID of the NAO result and also the name of the command that is used to examine the NAO and make changes to it. Such a command is called an *Object-Oriented Command (OOC)*. The ID is called the *OOC-name*. Continuing the above example, let's execute the OOC by typing its name '`::NAP::16-16`':

```
% ::NAP::16-16
7
```

This displays the value in the NAO.

However the following attempt to repeat the command fails because the NAO and its associated OOC were automatically deleted at the end of the first execution of the OOC. Nap detected the fact that this NAO was not referenced by anything and it was therefore treated as *use once and then discard*.

```
% ::NAP::16-16
invalid command name "::NAP::16-16"
```

Tcl syntax allows a command to include another command within square brackets `[]`. First the bracketed command is executed and its Tcl result replaces it. Then the modified whole command is executed. So the above commands can be simplified by enclosing the `nap` command in brackets as follows:

```
% [nap "2 * 3.5"]
7
```

Nap expressions can contain the assignment operator '`=`' with a Tcl variable name on its left and any expression on its right. For example:

```
% nap "result = 2 * 3.5"
::NAP::24-24
```

The following shows that this sets the Tcl variable `result` to the string value '`::NAP::24-24`'.

```
% set result
::NAP::24-24
```

Tcl syntax replaces '`$` *name*' by the contents of variable *name*. So in our example '`$result`' is replaced by '`::NAP::24-24`', as shown in the following:

```
% $result
7
```

The fact that this NAO is referenced by something (the variable '`result`') means that it is not deleted after it executes. So we can repeat the command:

```
% $result
7
```

We can also use variable names within Nap expressions. For example:

```
% [nap "result + 4"]
11
```

Note that no '$' is needed before a variable name in a Nap expression. Variables can also contain numeric strings, as in:

```
% set offset 8.2
8.2
% [nap "result + offset"]
15.2
```

An OOC can have arguments. The following example demonstrates the argument `all`, which requests additional information about the NAO.

```
% $result all
::NAP::24-24  f64  MissingValue: NaN  References: 1
Value:
7
```

The following additional information is provided:

- *OOC-name* `::NAP::24-24`
- *Data-type* `f64` (64-bit floating-point)
- *Missing-value* `NaN` (special value for missing data)
- *Reference-count* The value is `1` because there is one variable (`result`) pointing to this NAO. If it were 0 the NAO would be deleted.

4.3 Arguments of `nap` Command

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.

For example, the following works without quotes:

```
% [nap 2 * 3]
6
```

But the following fails without quotes:

```
% [nap {3 5} * 2]
3 5 * 2
  ^
syntax error, unexpected UNUMBER, expecting $end
Error at line 1768 of file napParse.tab.c
% [nap "{3 5} * 2"]
6 10
```

Multi-line expressions are convenient for matrices, as in:

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

4.4 Data Types

4.4.1 Nap Data-types

A NAO can have any of the following data-types:

Name	Description	Minimum	Maximum	Default Missing Value
`c8`	8-bit character	0	255	0
`i8`	8-bit signed integer	-128	127	-128
`i16`	16-bit signed integer	-32768	32767	-32768
`i32`	32-bit signed integer	-2147483648	2147483647	-2147483648
`u8`	8-bit unsigned integer	0	255	none
`u16`	16-bit unsigned integer	0	65535	65535
`u32`	32-bit unsigned integer	0	4294967295	4294967295
`f32`	32-bit floating-point	$-\infty$	∞	NaN
`f64`	64-bit floating-point	$-\infty$	∞	NaN
`ragged`	slot numbers	0	2147483647	0
`boxed`	slot numbers	0	2147483647	0

The `ragged` type provides an efficient way of storing arrays with many missing values around the edges.

A `boxed` NAO contains slot numbers pointing to other NAOs. This allows one to construct arrays (normally vectors) of arrays. Boxed vectors are generated by the *link operators* '`...`' and '`,`' described in section 5.3.3. Boxed vectors are used

- to pass multiple arguments to a function
- to return multiple results from a function
- in *cross-product indexing*
- to generate *arithmetic progressions* with step sizes other than 1 and -1

Boxed vectors can be unpacked using function `open_box(`x`)`, which is described in section 5.5.12 (Functions related to Special Data-types).

4.4.2 Data-type of Constants

A scalar constant can contain a data-type suffix, as in:

```
% [nap "3.7f32"] all
::NAP::53-53  f32  MissingValue: NaN  References: 0
Value:
3.7
% [nap "123u8"] all
::NAP::55-55  u8  MissingValue: (NULL)  References: 0
Value:
123
```

The following examples show that if there is no such suffix then floating-point values are treated as `f64` while integer values are treated as `i32`:

```
% [nap "3.7"] all
::NAP::57-57  f64  MissingValue: NaN  References: 0
Value:
3.7
% [nap "123"] all
::NAP::58-58  i32  MissingValue: -2147483648  References: 0
Value:
123
```

4.4.3 Data-type Conversion Functions

There is a function with the name of each data-type except `ragged` and `boxed`. Each such function converts its argument to that data-type. For example:

```
% [nap "f64(123)"] all
::NAP::62-62  f64  MissingValue: NaN  References: 0
Value:
123
% [nap "c8(123)"] all; # ascii character 123 (left-brace i.e. '{')
::NAP::66-66  c8  MissingValue: (NULL)  References: 0
Value:
{
% [nap "f32({123 -1.2 0})"] all; # vector with 3 elements
::NAP::70-70  f32  MissingValue: NaN  References: 0
Dimension 0   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
123 -1.2 0
```

The parentheses `()` in these three examples are not needed. Deleting them:

```
% [nap "f64 123"] all
::NAP::26-26  f64  MissingValue: NaN  References: 0
Value:
123
% [nap "c8 123"] all
::NAP::29-29  c8  MissingValue: (NULL)  References: 0
Value:
{
% [nap "f32{123 -1.2 0}"] all
::NAP::33-33  f32  MissingValue: NaN  References: 0
Dimension 0   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
123 -1.2 0
```

4.4.4 Data-type of result of operation

Many operations (defined by an operator or a function) produce a result whose data-type matches that of their operands/arguments (if these all have the same data-type). The following examples illustrate this for the *subtract* operator:

```
% [nap "3 - 1"] all
::NAP::57-57  i32  MissingValue: -2147483648  References: 0  Unit: (NULL)
Value:
2
% [nap "3f32 - 1f32"] all
::NAP::61-61  f32  MissingValue: NaN  References: 0  Unit: (NULL)
Value:
2
% [nap "3u8 - 1u8"] all
::NAP::65-65  u8  MissingValue: 255  References: 0  Unit: (NULL)
Value:
2
```

What happens if the operands differ in data-type? Let's try adding `f32` and `f64` values:

```
% [nap "234f32 + 3.5f64"] all
::NAP::40-40  f64  MissingValue: NaN  References: 0
Value:
237.5
```

The result is of type `f64`, so there is no loss of precision. Next let's try adding `i32` and `f32` values:

```
% [nap "234 + 3.5f32"] all
::NAP::54-54  f64  MissingValue: NaN  References: 0  Unit: (NULL)
Value:
237.5
```

Why is the result `f64` rather than `f32`? This prevents possible loss of precision, as in:

```
% [nap "f32(123456789) + 5f32"] all -format %d
::NAP::48-48  f32  MissingValue: NaN  References: 0  Unit: (NULL)
Value:
123456800
```

This is due to the fact that an `i32` value has 31 bits (9.3 digits) of precision, whereas an `f32` value has only 24 bits (7.2 digits) of precision. So both operands must be converted to `f64` before the addition takes place. This is shown by:

```
% [nap "123456789 + 5f32"] all -format %d
::NAP::43-43  f64  MissingValue: NaN  References: 0  Unit: (NULL)
Value:
123456794
```

Some operations produce a result whose data-type is independent of the types of the operands. In particular, relational and logical operators always produce an `i8` result with value 1 for true and 0 for false. The following illustrates the '>' (greater than) operator:

```
% [nap "9 > 8"] all
::NAP::43-43  i8  MissingValue: (NULL)  References: 0
Value:
1
```

4.5 Arrays

4.5.1 Introduction

Nap stores data in *NAOs*. A NAO is essentially an object based on the concept of an *n-dimensional array*. The number of dimensions is called the *rank*. A scalar is simply an array of rank 0.

Dimensions can have names and *coordinate variables (CVs)*. A CV maps a subscript to another variable such as a physical dimension (e.g. time). Arrays with CVs are called *grids*.

Nap provides powerful and efficient facilities for processing arrays, including:

- *elemental* (element by element) extension of scalar arithmetic
- array operations such as sum, concatenation and reshaping
- indexing in a variety of ways
- searching (inverse indexing)

4.5.2 Vectors

An array with exactly one dimension is called a *vector*.

Nap array constants are enclosed in braces ('{}'). The following example doubles a vector containing four elements:

```
% [nap "2 * {3.4 -0.1 _ 7}"]
6.8 -0.2 _ 14
```

Note that an underscore ('_') represents a missing element.

The ordinary scalar arithmetic operators and functions are applied to arrays in an elemental fashion, as shown by:

```
% [nap "{1 _ 4 -6} + {2 0 -9 _}"]
3 _ -5 _
% [nap "sqrt(-{1 _ 9 -6} + 10)"]
3 _ 1 4
```

The function `sum` has the functionality of the mathematical '$\sum$'. When applied to a vector it returns the sum of the elements (ignoring any that are missing), as shown by:

```
% [nap "sum{1 _ 9 -6 3}"]
7
```

The following example applies the concatenation operator '`//`' to vectors and scalars:

```
% [nap "{1 _ 9} // 7 // {-3 8}"]
1 _ 9 7 -3 8
```

The operators '`..`' and '`...`' generate *arithmetic progression (AP)* vectors. For example:

```
% [nap "3 .. 7"]
3 4 5 6 7
% [nap "7 .. 3"]
7 6 5 4 3
% [nap "0 .. 10 ... 2.5"]; # From 0 to 10 with step of 2.5
0 2.5 5 7.5 10
% [nap "5 ... 0 .. 10"]; # 5 elements from 0 to 10
0 2.5 5 7.5 10
```

An *index* specifies a position within an array, commencing from 0. If two operands (representing NAOs) are adjacent then Nap treats the right operand as the index of the left operand. For example:

```
% nap "abc = {4 9 8 2}"
::NAP::283-283
% [nap "abc(2)"]
8
% [nap "abc 0"]; # parentheses not needed
4
% [nap "abc{2 0 3 2 1}"]; # vector index
8 4 2 8 9
```

Some computer languages allow indexed variable names on the left of the assignment operator '`=`'. So one might expect to be able to modify element 2 of `abc` as follows:

```
% nap "abc(2) = 5"
abc(2) = 5
       ^
syntax error, unexpected '=', expecting $end
Error at line 1768 of file napParse.tab.c
```

Nap does not allow indexed variable names on the left of the assignment operator '`=`'. However the '`set value`' OOC method does provide a way to modify elements of a NAO. Element 2 of `abc` can be changed to 5 as follows:

```
% $abc set value 5 2
% $abc
4 9 5 2
```

The following examples show how several elements (or the entire array) can be changed using a single OOC:

```
% $abc set value "{13 11}" "{0 2}"
% $abc
13 9 11 2
% $abc set value -3
% $abc
-3 -3 -3 -3
% $abc set value "{13 11}"
% $abc
13 11 13 11
```

The default OOC displays only the first six elements, as in:

```
% [nap "1 .. 9"]
1 2 3 4 5 6 ..
```

The '`value`' OOC method displays all elements, as in:

```
% [nap "1 .. 9"] value
1 2 3 4 5 6 7 8 9
```

4.5.3 Matrices

An array with exactly two dimensions is called a *matrix*. A matrix constant has two levels of braces, as in:

```
% [nap "{{2 4 8}{0 -1 9}}"] all
::NAP::298-298  i32  MissingValue: -2147483648  References: 0
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
 2  4  8
 0 -1  9
```

It is often convenient to use a separate line for each row, as in:

```
% nap "mat = {
{2 1.5 7}
{0  -2 1.1}
}"
::NAP::301-301
% $mat all
::NAP::301-301  f64  MissingValue: NaN  References: 1
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
 2.0  1.5  7.0
 0.0 -2.0  1.1
```

The function '`sum`' gives column and row sums, as in:

```
% [nap "sum(mat)"]
2 -0.5 8.1
% [nap "sum(mat, 1)"]
10.5 -0.9
```

Matrices (and higher rank arrays) can be indexed using either *cross-product indexing* or *full indexing*. A cross-product index of a matrix has the form '*rows*, *columns*', where *rows* and *columns* are scalars or vectors. Here are two examples of cross-product indexing:

```
% [nap "mat(0,2)"]
7
% [nap "mat(0 .. 1, {0 2})"]
2.0 7.0
0.0 1.1
```

A full index of a matrix is an array with two columns corresponding to *row* and *column*. Here are two examples of full indexing:

```
% [nap "mat{0 2}"]
7
% [nap "mat{{0 2}{0 0}{1 3}}"]
7 2 0
```

The function '`transpose`' swaps the row and column dimensions. E.g.

```
% $mat
 2.0  1.5  7.0
 0.0 -2.0  1.1
% [nap "transpose mat"]
 2.0  0.0
 1.5 -2.0
 7.0  1.1
```

The function '`reshape`' uses the elements of the first argument (recycled if necessary) to produce an array whose shape is specified by the second argument. If there is only one argument then this is reshaped to a vector with the same number of elements. E.g.

```
% [nap "reshape(mat, {3 4})"]
 2.0  1.5  7.0  0.0
-2.0  1.1  2.0  1.5
 7.0  0.0 -2.0  1.1
% [nap "reshape mat"]
2 1.5 7 0 -2 1.1
% [nap "reshape(9, {2 5})"]
9 9 9 9 9
9 9 9 9 9
```

The concatenation operator '`//`' can be applied to matrices. For example:

```
nap "m33 = {{4 6 8}{3 5 9}{0 4 1}}"
::NAP::14-14
% $m33
4 6 8
3 5 9
0 4 1
% $mat
 2.0  1.5  7.0
 0.0 -2.0  1.1
% [nap "m33 // mat"]
 4.0  6.0  8.0
 3.0  5.0  9.0
 0.0  4.0  1.0
 2.0  1.5  7.0
 0.0 -2.0  1.1
```

The laminate operator '`///`' joins its operands over a new dimension. The following example creates a matrix from two vectors:

```
% [nap "{9 3 5} /// {1 7 2}"]
9 3 5
1 7 2
% [nap "{9 3 5} // {1 7 2}"]; # Note difference
9 3 5 1 7 2
```

4.5.4 Higher Rank Arrays

A NAO can have up to 32 dimensions. The following example creates a 3-dimensional array, sums it along each dimension, then extracts an element using cross-product indexing.

```
% nap "a3D = reshape(0 .. 99, {2 3 4})"
::NAP::47-47
% $a3D
```

```
 0  1  2  3
 4  5  6  7
 8  9 10 11

12 13 14 15
16 17 18 19
20 21 22 23
% [nap "sum(a3D)"]
12 14 16 18
20 22 24 26
28 30 32 34
% [nap "sum(a3D, 1)"]
 6 22 38
54 70 86
% [nap "sum(a3D, 2)"]
12 15 18 21
48 51 54 57
% [nap "a3D(0,2,1)"]
9
```

4.5.5 Grids

Grids are arrays with *coordinate variables (CVs)*. Dimensions often correspond to physical dimensions such as time and length. A CV maps a subscript to another variable, which is typically a physical dimension such as time.

The following example creates a NAO corresponding to the grid described in section 2.4.3 (Missing Data). Note how the '`set coo`' OOC method is used to attach the CVs and name the dimensions.

```
% nap "latitude = {30 25 10 -10}"
::NAP::74-74
% nap "longitude = {30 40 60 65 75}"
::NAP::76-76
% nap "grid2d = {
{15 17 10  _ 21}
{16 14 18 18 19}
{ _ 11 12 11  _}
{10  9  _ 12 11}
}"
::NAP::78-78
% $grid2d set coo latitude longitude
% $grid2d all
::NAP::78-78  i32  MissingValue: -2147483648  References: 1
Dimension 0   Size: 4      Name: latitude  Coordinate-variable: ::NAP::74-74
Dimension 1   Size: 5      Name: longitude  Coordinate-variable: ::NAP::76-76
Value:
15 17 10  _ 21
16 14 18 18 19
 _ 11 12 11  _
10  9  _ 12 11
```

Grids can be indexed using *indirect indexing*, which is indexing via CVs. Continuing the above example, let us extract the value corresponding to a latitude of 25 and a longitude of 75:

```
% [nap "grid2d(@25, @75)"]; # indirect indexing
19
% [nap "grid2d(1, 4)"]; # direct indexing
19
```

Nap allows fractional subscripts, which correspond to interpolated values. The following produces a value corresponding to row 0.5 (27.5°N) and column 1.5 (50°E):

```
% [nap "grid2d(0.5, 1.5)"]; # direct indexing
14.75
% [nap "grid2d(@27.5, @50)"]; # indirect indexing
14.75
% expr (17+10+14+18) * 0.25; # check arithmetic
14.75
```

4.5.6 Strings

String constants are enclosed in apostrophes ‘’’ or grave accents ‘‘‘ and generate character vectors. For example:

```
% nap "message = 'hello world'"
::NAP::329-329
% $message all
::NAP::329-329  c8  MissingValue: (NULL)  References: 1
Dimension 0   Size: 11     Name: (NULL)    Coordinate-variable: (NULL)
Value:
hello world
% [nap "‘don't‘"]
don't
```

The following example uses the library function ‘gets_matrix’ to read the text file ‘abc.txt’ into a NAO referenced by ‘in’.

```
% nap "in = gets_matrix('abc.txt')"
::NAP::349-349
% $in all
::NAP::349-349  f64  MissingValue: NaN  References: 1
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
 2  9 -3
 0 12  7
```

Chapter 5

Nap Expressions

5.1 Syntax of Nap Expressions

5.1.1 Introduction

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 ('`[]$`')

5.1.2 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.

Similarly one can use a Tcl variable as a generic function, as in:

```
% set f sqrt
sqrt
% [nap "f 9"]
3
```

5.1.3 Nap Names

Nap names are used to identify variables and functions. Nap variables are just Tcl variables pointing to NAOs. Tcl procedures can be defined to be called as Nap functions. Nap does allow names to be prefixed by namespace pathnames.

Tcl procedures have always had their own namespaces for the names of variables. However many packages need to use many global names for both variables and procedures. Namespaces (introduced into Tcl in version 8.0) provide a systematic heirarchical (tree) structure for such global names, just as file directories (folders) provide such a structure for files. The components of a Tcl name are separated by a double colon (‘::’) separator.

The following examples illustrate the use of namespaces with Nap.

```
% namespace eval ::mySpace {}; # create namespace "mySpace"
% nap "::mySpace::x = 8"
::NAP::13-13
% [nap "3 + ::mySpace::x"]
11
% proc ::mySpace::square x {nap "x * x"}; # Define function
% [nap "::mySpace::square {3 5 2}"];      # Call it
9 25 4
```

Tcl allows any string to be the name of a variable or a procedure. The rules for Nap names are more restrictive. The only valid characters are letters, digits, underscore (‘_’) and colons. Colons are only allowed as double colon (‘::’) namespace separators. The final component (excluding the namespace pathname) must commence with either a letter or an underscore. If it commences with an underscore it must contain at least one other character. The components of the namespace pathname must either conform to the same rule or be unsigned integers.

5.1.4 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 an operand (now an 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
```

This 'sin' function (and 'hypot' used below) are described in section 5.5.1.

Multiple arguments are separated by commas in the usual way. For example:

```
% [nap "hypot(3,4)"]
5
```

A comma is treated as a low-precedence operator which produces a *boxed* array. The parentheses *are* needed to force the comma to be executed before the function. The fact that comma is just another operator is shown by:

```
% nap "a = 3"
::NAP::13-13
% nap "b = 4"
::NAP::14-14
% nap "both = a , b"
::NAP::15-15
% $both all
::NAP::15-15  boxed  MissingValue: 0  References: 1
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Value:
13 14
% [nap "hypot both"]
5
```

Note that the values of '`both`' are the slot numbers of '`a`' and '`b`' .

5.1.5 Indexing

Tcl array indices are enclosed by parentheses ('`()`'), while C uses brackets ('`[]`'). Nap requires neither, since indexing is simply implied by adjacent operands (now 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
```

5.1.6 Parenthesising Function Arguments and Indices

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.

Multiple function arguments and cross-product-indexing are both defined using a *boxed* argument. A boxed argument is normally produced using the comma operator. The comma operator has low precedence, so parentheses *are* normally needed for multiple function arguments and cross-product-indexing. The following example shows how an element of a matrix can be extracted using either *full* or *cross-product* indexing:

```
% nap "matrix = {
{2 3 4}
{5 6 7}
}"
::NAP::30-30
% [nap "matrix{0 2}"]; # Full index
4
% [nap "matrix(0,2)"]; # Cross-product index
4
```

5.2 Constants

5.2.1 Introduction

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.

5.2.2 Integer Scalar Constants

An integer scalar constant can be specified in decimal or hexadecimal form. The default data-type is `i32` (32-bit signed integer) for decimal integer constants. Octal constants are not allowed from version 3, although they were in earlier versions (causing problems with decimal data containing leading 0s).

Hexadecimal integer constants begin with '`0x`' and are 32-bit unsigned integers. A data-type suffix is not allowed for hexadecimal constants because some cases would be ambiguous.

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 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.

5.2.3 Floating-point Scalar Constants

A floating-point scalar constant can represent infinity, 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 *scalar* decimal mantissa must not begin or end with a decimal point. (However this rule is relaxed within array constants as discussed below.) 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 1e]; # Default exponent is 1
10
% [nap 1p1]
3.14159
% [nap 1p]; # Default exponent is 1
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:
-
```

5.2.4 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 except that *array* floating point mantissas can begin or end with a decimal point.

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

```
% [nap "{2 .8 -7.}"] all
::NAP::16-16  f64  MissingValue: NaN  References: 0
Dimension 0   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
2 0.8 -7
```

Note that '`.8`' begins with a decimal point and '`-7.`' ends with one.

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
```

Elements can be preceded by a '+' or '-' sign. 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
```

5.2.5 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
```

5.3 Operators

5.3.1 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 operands is indicated as follows:
a and b represent general arrays.
x, y and z represent scalars.
u and v represent vectors.
A and B represent matrices.
i represents an integer scalar subscript
s represents a real (possibly fractional) scalar subscript
n represents a name, which may include namespaces.
p represents a boxed vector of pointers to arrays a_0, a_1, a_2, ...

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	$\|a\|$ (Absolute value of a)
`^` a	Nearest integer to a
$<a$	$\lfloor a \rfloor$ (Largest integer not greater than a)
$>a$	$\lceil a \rceil$ (Smallest integer not less than a)
`~` a	Bit-wise complement of a
`#` a	Frequencies of values 0, 1, 2, ...
`@` a	Indirect subscript
`@@` a	Indirect subscript
`<=` u	Permutation vector v such that u_v is in ascending order
`>=` u	Permutation vector v such that u_v is in descending order
v `@` b	Real (possibly fractional) s for which $v_s = b$
v `@@` b	Integer i for which $\|v_i - b\|$ is least
v `@@@` b	Smallest integer i for which $v_i = b$
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	a mod 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 $a \leq b$, else 0
a `>=` b	1 if $a \geq b$, 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 (inclusive) OR of a and b
a `&&` b	Logical AND: 1 if $a \neq 0$ and $b \neq 0$, else 0
a `||` b	Logical (inclusive) OR: 1 if $a \neq 0$ or $b \neq 0$, else 0
x `..` y	Arithmetic Progression from x to y in steps of `+1` or `-1`
x `..` y `...` z	Arithmetic Progression from x to y in steps of z
x `...` y `..` z	Arithmetic Progression from y to z with x elements
a `?` b `:` c	Choice: if $a \neq 0$ then b, else c
a `//` b	Concatenate along existing dimension
a `///` b	Concatenate along new dimension
[a]`,`[b]	Boxed vector pointing to a and b (unless already boxed) (If a or b is already boxed then concatenate it)
n `=` a	Result is a. Right-associative. Side Effect: Set n to OOC-name of a

5.3.2 Assignment Operator '='

The '`nap`' command (unlike '`expr`') allows the assignment operator '`=`'.

The left-hand operand must be a name, as defined in section 5.1.3. This is used as the name of a Tcl variable (which may or may not already exist) 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
```

5.3.3 Link Operator ‘,’

The link operator ‘,’ produces 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 section 5.4.2.4.

The left-hand operand of ‘,’ generates one boxed vector and the right-hand 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.

5.3.4 Arithmetic Progression Operators ‘..’ and ‘...’

The operator ‘..’ generates an *arithmetic progression* (AP). If both operands are simple numeric scalars then the step size is $+1$ or -1, the left-hand operand specifies the first value and the right-hand 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-hand 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-hand 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-hand 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
```

5.3.5 Concatenation Operators '//' and '///'

The following example illustrates the difference between '//' and '///' with vector operands:

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

The following example illustrates the difference between '//' and '///' with matrix operands:

```
% [nap "{{6 2 1}{0 9 4}} // {{7 2 7}{3 3 8}}"] all
::NAP::29-29  i32  MissingValue: -2147483648  References: 0  Unit: (NULL)
Dimension 0   Size: 4      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
6 2 1
0 9 4
7 2 7
3 3 8
% [nap "{{6 2 1}{0 9 4}} /// {{7 2 7}{3 3 8}}"] all
::NAP::35-35  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:
6 2 1
0 9 4
```

```
7 2 7
3 3 8
```

Note that '//' concatenates along the most significant existing dimension, whereas '///' concatenates along a new dimension. This new dimension is of size 2 and is more significant that the existing dimensions.

The above examples had operands with identical shapes and data-types. It is obviously desirable to allow the operands of '//' to have different sized leading (most significant) dimensions. Nap does allow this, as shown by:

```
% [nap "'Hello' // ' world.'"]
Hello world.
% [nap "{{6 2 1}{0 9 4}} // {{7 2 7}}"]
6 2 1
0 9 4
7 2 7
```

In fact, both operators allow any combination of shapes. Operands of '///' are reshaped to the same shape. Operands of '//' are reshaped so all dimensions except the leading one have the same size. The following examples illustrate this reshaping process (with data-type conversion when required):

```
% [nap "{{6 2 1}{0 9 4}} // {7 2 7}"]
6 2 1
0 9 4
7 2 7
% [nap "{{6 2 1}{0 9 4}} // 3.0"] all
::NAP::142-142  f64  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
6 2 1
0 9 4
3 3 3
% [nap "{{6 2 1}{0 9 4}} /// 3.0"] all
::NAP::148-148  f64  MissingValue: NaN  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:
6 2 1
0 9 4

3 3 3
3 3 3
```

5.3.6 Inverse Indexing Operators '@', '@@' and '@@@'

These three operators all take an optional vector left-hand operand. (The '@' operator also allows the left-hand operand to have a rank greater than 1.) The result is a subscript of this vector. The left-hand operand defaults to the coordinate-variable of the dimension (only relevant to *indirect indexing*, which is discussed in section 5.4.3).

The right-hand operand is attached to the result using its *link slot* (see section 13.1). This enables the right-hand operand to be automatically used as a coordinate variable if the result is directly used as an index. Note that the results of operators/functions do not normally retain any links in their operands/arguments, so this only applies to *direct* use. (The right-hand operand would not be an appropriate coordinate variable if there were further arithmetic prior to indexing.)

5.3.6.1 Interpolated Subscript '@'

The result of 'v@b' is a real (possibly fractional) `f32` subscript value s such that $v_s = b$.

If v is strictly monotonic (sorted without duplicates) then there is only one value of s for which $v_s = b$. This unique value is obviously the result.

If v is monotonic with duplicates then there can be several duplicate elements exactly matching b. In this case the result is the mean of the subscripts of such matching elements.

If v is not monotonic then the result is the smallest real (possibly fractional) `f32` subscript value s such that $v_s = b$.

Let us begin with some strictly monotonic examples:

```
% [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 quarter-way 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
```

The following example shows how extrapolation is used to define the result when the right-hand operand is outside the range of the left-hand operand:

```
% [nap "{-1 0 2} @ {-2 5}"]
-1 3.5
```

Such extrapolation can be prevented by adding end points with missing or infinite values, as in:

```
% [nap "{_ -1 0 2 _} @ {-2 -1 2 5}"]
_ 1 3 _
% [nap "{-1i -1 0 2 1i} @ {-2 -1 2 5}"]
1 1 3 3
```

The effect of other missing values is shown by:

```
% [nap "{_ 2 4 _ 6 8 _} @ (1 .. 9)"] value
_ 1 1.5 2 _ 4 4.5 5 _
```

Now let us consider an example where the left operand is monotonic with duplicates and there are multiple exact matches. Note how the result is defined as the mean of the matching subscripts.

```
% [nap "{1.3 6.5 6.5 7.1} @ 6.5"]
1.5
```

Next let us consider an example where the left operand is not monotonic (sorted) at all. Note how the result is defined by the first match, which need not be exact.

```
% [nap "{2 4 5 3} @ (1 .. 6)"]
-0.5 0 0.5 1 2 _
```

The left-hand operand can have a rank greater than 1. In this case the search takes place over the most significant dimension (0) of the left-hand operand. The following example searches down each column for the value `0.7`.

```
% nap "mat = {
    {0.3 0.1 0.9}
    {0.5 0.5 0.8}
    {0.6 0.1 0.6}
    {0.8 0.0   _}
}'
::NAP::157-157
% [nap "mat @ 0.7"]
2.5 _ 1.5
```

Thus this combines the effect of the following three commands.

```
% [nap "{0.3 0.5 0.6 0.8} @ 0.7"]
2.5
% [nap "{0.1 0.5 0.1 0.0} @ 0.7"]
_
% [nap "{0.9 0.8 0.6 _} @ 0.7"]
1.5
```

The right-hand operand can have any rank, but trailing dimensions (excluding dimension 0 of the left-hand operand) must match. The following example has a right-hand operand with the same number (3) of columns as `mat`.

```
% [nap "mat @ {{0.7 0.7 0.7}{0.4 0.5 0.8}}"]
2.5   _ 1.5
0.5 1.0 1.0
```

The following 3D array contains ocean temperature data for 4 depths, 2 latitudes and 3 longitudes. Note that some (shallower) points have missing values at the deepest level. For each (latitude, longitude) point, we want to find the depth (subscript) corresponding to a temperature of 10 degrees. The missing value in the result corresponds to an oceanic column whose minimum temperature is 12.

```
% nap "temperature = {
    {{11 12 13}{11 11 12}}
    {{ 9  9 13}{11  8 10}}
    {{ 8 10 12}{ 9  8 10}}
    {{ 6  2  _}{ 5  _  _}}
}'
% [nap "temperature @ 10"]
0.500000 0.666667        _
1.500000 0.333333 1.000000
```

5.3.6.2 Subscript of Closest '@@'

The result of 'v`@@`b' is the `i32` subscript s for which $|v_s - b|$ 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.

The following example shows how the right-hand operand becomes the coordinate variable if the result is used directly as an index, but not if there is further arithmetic.

```
% nap "coarse = {4 8 7}"
::NAP::14-14
% nap "time = {2 3 5}"
::NAP::16-16
```

```
% $coarse set coo time
% [nap "fine = coarse(time@@(2.4 .. 4.6 ... 0.2))"] value
4 8 8 8 8 8 8 8 7 7 7
% [$fine coo] value; # Display coordinate variable
2.4 2.6 2.8 3 3.2 3.4 3.6 3.8 4 4.2 4.4 4.6
% [nap "fine = coarse(time@@(2.4 .. 4.6 ... 0.2)+1)"] value;
# Do further arithmetic
8 7 7 7 7 7 7 7 7 4 4 4
% [$fine coo] value; # Display coordinate variable
3 5 5 5 5 5 5 5 5 2 2 2
```

5.3.6.3 Subscript of Match '@@@'

The result of 'v@@@b' is the smallest `i32` subscript i for which $v_i = b$. 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
```

5.3.7 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 operand 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
```

If the operand is boxed and points to n arrays (which each have the same number of elements) then the result is the n-dimensional array of joint frequencies. For example:

```
% [nap "#({2 1 1 0 1},{1 1 3 2 1})"]
0 0 1 0
0 2 0 1
0 1 0 0
```

The boxed operand defines the five pairs (2,1), (1,1), (1,3), (0,2) and (1,1). The above result gives the frequencies of these pairs.

5.3.8 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-hand operand defines the number of replications of the corresponding element of the right-hand operand. The operands 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 operand is:

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

If the right-hand operand b is multidimensional then the left-hand operand must be a boxed vector pointing to vectors corresponding to the dimensions 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
```

5.3.9 Remainder Operator '%'

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

5.3.10 Unary Sorting Operators '<=' and '>='

These operators are applied to vectors to produce the permutation vector which (when applied as its index) sorts the argument into ascending or descending order.

Thus '$u = v$`(<=`v`)`' is sorted into ascending order, so that $u_0 \leq u_1 \leq u_2 \leq \ldots$
and '$u = v$`(>=`v`)`' is sorted into descending order, so that $u_0 \geq u_1 \geq u_2 \geq \ldots$

The floating-point value *NaN* is treated as less than $-\infty$. The current version treats other missing values as having their numeric value. (This may change in future versions.)

The following examples illustrate these two operators and the related function `sort()` (see section 5.5.6):

```
% [nap "x = {1.5 -1i 0 _ 9 _ 0 1i -2 1}"] value
1.5 -Inf 0 _ 9 _ 0 Inf -2 1
% [nap "pv = <= x"] value; # permutation vector
3 5 1 8 6 2 9 0 4 7
% [nap "x(pv)"] value; # ascending order
_ _ -Inf -2 0 0 1 1.5 9 Inf
% [nap "sort(x)"] value; # same result
_ _ -Inf -2 0 0 1 1.5 9 Inf
% [nap "x(>= x)"] value; # descending order
Inf 9 1.5 1 0 0 -2 -Inf _ _
```

5.4 Indexing

5.4.1 Introduction

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 argument that 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
```

5.4.1.1 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 parenthesise an index that 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
```

5.4.1.2 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 $0.5\times9.0+0.5\times4.0 = 4.5+2.0 = 6.5$ using ordinary one-dimensional linear interpolation.

If n is the dimension-size and p the position, then $0 \leq p < n$. Values between n-1 and n are defined by treating position n 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 $0.9 \times 4.0 + 0.1 \times 2.0 = 3.6 + 0.2 = 3.8$

5.4.1.3 Subscript

Dimension-positions are always specified via *subscripts*. A *subscript* is similar to a *dimension-position* except that there are no size limits. If s is the subscript and n is the dimension-size, then the dimension-position p is defined by s `%` n, the remainder after dividing s by n.

Thus in in the following example subscript `6` is treated as `6%4` = `2`.

```
% [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
```

5.4.1.4 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 $0.25 \times 0.0 + 0.25 \times 7.0 + 0.25 \times (-4.0) + 0.25 \times (-9.0) = -1.5$

5.4.2 Index

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

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

5.4.2.1 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
```

5.4.2.2 Vector-Flip

It is often necessary to reverse the order of elements in a vector. One could use *shape-preserving* indexing, as in:

```
% [nap "{2 4 6 8}(3 .. 0)"]
8 6 4 2
```

Nap provides the *niladic* operator '-' to specify such reversal (or *flipping*). (A *niladic* operator is one without any operands.) Thus one can simplify the above example to:

```
% [nap "{2 4 6 8}(-)"]
8 6 4 2
```

Such an index of a vector, consisting of just '-', is called a *vector-flip*. Note that cross-product-indexing (see section 5.4.2.4) also allows the niladic '-' to specify flipping of one or more dimensions.

What does the niladic '-' generate? Let's see:

```
% [nap "-"] all
::NAP::62-62  f32  MissingValue: NaN  References: 0  Unit: (NULL)
Value:
-Inf
```

It generates a scalar 32-bit NAO with the value $-\infty$. Indexing treats such a NAO as meaning 'flip'. So the above indexing example could also (but less conveniently) be written as:

```
% [nap "{2 4 6 8}(-1if32)"]
8 6 4 2
```

5.4.2.3 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.

5.4.2.4 Cross-product-index

A *cross-product-index* is a boxed vector containing *rank* elements pointing to scalars, vectors, nulls and flips. (Flips are discussed in section 5.4.2.2). The cross-product combination of this vector defines the elemental indices 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..(`*n*`-1)`', where *n* is the dimension-size. Scalar operands produce no corresponding dimension in the result. A flip (dimension reversal) is normally represented by the niladic '`-`' operator, which is equivalent to '`(`*n*`-1)..0`'.

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; # full-index
::NAP::196-196  f32  MissingValue: NaN  References: 0  Unit: (NULL)
Value:
-1.5
% [nap "mat(0.5,1.5)"] all; # cross-product-index
::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; # cross-product-index
::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; # full-index
::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 examples show how the niladic ‘-’ operator is used to flip (reverse) dimensions:

```
% $mat
 1.5  0.0  7.0
 2.0 -4.0 -9.0
% [nap "mat(,-)"]
 7.0  0.0  1.5
-9.0 -4.0  2.0
% [nap "mat(-,)"]
 2.0 -4.0 -9.0
 1.5  0.0  7.0
% [nap "mat(-,-)"]
-9.0 -4.0  2.0
 7.0  0.0  1.5
% [nap "mat(0,-)"]
7 0 1.5
% [nap "mat(-,{2 0 0})"]
-9.0  2.0  2.0
 7.0  1.5  1.5
```

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
```

5.4.3 Indirect Indexing and Unary Operators '@' and '@@'

Indirect indexing is indexing via coordinate variables. An example is specifying latitudes and longitudes rather than rows and columns. One can convert a latitude of say 20°S to a subscript using '`lat@-20`', where `lat` is the latitude coordinate variable. This uses the *binary* interpolated subscript '`@`' operator discussed in section 5.3.6.1. Within an index expression it is possible to use the *unary* '`@`' operator and simply write '`@-20`'. The omitted left operand defaults to the corresponding coordinate variable.

There are three such unary operators for indirect indexing: '`@`', '`@@`' and '`@@@`'. However '`@@@`' is seldom used and will not be mentioned further.

There were restrictions on the use of these unary operators prior to version 5 of nap. Each subscript had to have the form '`@`*expr*' or '`@@`*expr*'. The unary '`@`' and '`@@`' operators worked by simply creating a copy of their operand and attaching to it (via the *link slot* discussed in section 13.1) an ancillary NAO containing an integer with the value

- 1 for indirect indexing using '`@`'
- 2 for indirect indexing using '`@@`'.

The indexing code still handles such indices. They are still produced by function `invert_grid()` (see section 5.5.10).

Version 5 introduced a parser which produces a *parse tree* and then executes this tree. This allows cleverer execution of various things such as indexing, during which the coordinate variables are visible and can be used by the unary '`@`' and '`@@`' operators. This allowed the above restrictions to be lifted, as demonstrated at the end of the following *1D Time-Series Example* in section 5.4.3.1.

However version 5 did not allow these unary operators to be used with *full indexing*. This problem is fixed in version 6, which treats arguments with *rank* > 1 in the old manner described above.

The unary operators '`@`' and '`@@`' are often used with indexing to interpolate to a finer or coarser grid. In this case the operand of '`@`' and '`@@`' is normally the desired coordinate variable of the result. In many cases, normal nap processes would result in this operand being used for the coordinate variable, but there are some situations where this would not be the case. One example

is producing a finer grid using '@@' to produce *nearest-neighbour* values. Another is using '@' with longitude wrap-around.

Nap ensures that the right-hand operand is used as the coordinate variable by attaching it to the result of '@' and '@@' using the *link slot* (see section 13.1). This is discussed further in section 5.3.6. The following 2D Geographic Example (in section 5.4.3.2) shows how '@@' produces *nearest-neighbour* values and such coordinate variables.

5.4.3.1 1D Time-Series Example

Suppose we have temperatures at two-hourly intervals from time 10:00 to 16:00 as follows:

```
% nap "time = 10 .. 16 ... 2"
::NAP::20-20
% nap "temperature = {20.2 21.6 24.9 22.7}"
::NAP::21-21
% $temperature set coord time
```

We could estimate temperatures every hour during this period using either the binary or unary '@' as follows:

```
% [nap "temperature(time @ (10 .. 16))"] value; # Use binary @
20.2 20.9 21.6 23.25 24.9 23.8 22.7
% [nap "temperature(@ (10 .. 16))"] value; # Use unary @
20.2 20.9 21.6 23.25 24.9 23.8 22.7
```

These unary operators can only be used within index expressions. Let's see what happens if we use one elsewhere:

```
% nap "@10"
Nap_Indirect: Illegal coordinate variable corresponding to unary '@'
Error at line 583 of file
/cygdrive/c/dav480/tcl/nap/generic/napMonad.c,m4
expr1: Error with unary operator '@'
Error at line 52 of file c:/dav480/tcl/tcl-nap/generic/eval_tree.c
```

Note that this unary '@' operator makes no sense because it is not within an index and thus there is no corresponding coordinate variable.

The following examples illustrate useful index expressions which now work but did not work prior to version 5 of NAP:

```
% [nap "temperature(@@11.5 + 2)"]; # Example 1
22.7
% [nap "temperature(@@11.5 .. @@16.5)"]; # Example 2
21.6 24.9 22.7
% [nap "temperature(>@10.5 .. <@15.5)"]; # Example 3
21.6 24.9
```

Example 1 gives the second temperature after that closest to time 11:30. Example 2 gives all the temperatures from that closest to time 11:30 to that closest to time 16:30. Example 3 gives all the temperatures from the first following time 10:30 to the first before time 15:30.

5.4.3.2 2D Geographic Example

The following creates a 3×4 matrix `temperature`, which has

- unit of `degC` (°C).
- 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:

```
% $temperature all
::NAP::72-72  f32  MissingValue: NaN  References: 1  Unit: degC
Dimension 0   Size: 3      Name: latitude  Coordinate-variable: ::NAP::76-76
Dimension 1   Size: 4      Name: longitude  Coordinate-variable: ::NAP::86-86
Value:
31.5 37.2 32.9 34.0
25.1 25.2 29.0 21.9
20.5 21.2 21.0 19.9
% [$temperature coo 0] 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
% [$temperature coo 1] 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`

The above use of '`@`' produces *interpolated* values. The following illustrates the use of '`@@`' to produce *nearest-neighbour* values. Note the use of the original operands as the final coordinate variables.

```
% nap "nearest_temperature = temperature(@@(14 .. 16), @@(123 .. 127))"
::NAP::222-222
```

```
% $nearest_temperature all
::NAP::222-222  f32  MissingValue: NaN  References: 1  Unit: degC
Dimension 0   Size: 3      Name: latitude  Coordinate-variable: ::NAP::212-212
Dimension 1   Size: 5      Name: longitude  Coordinate-variable: ::NAP::217-217
Value:
37.2 37.2 32.9 32.9 32.9
25.2 25.2 29.0 29.0 29.0
25.2 25.2 29.0 29.0 29.0
% [$nearest_temperature coo 0]
14 15 16
% [$nearest_temperature coo 1]
123 124 125 126 127
```

5.5 Built-in Functions

5.5.1 Elemental Functions

An *elemental function* is a function with the following properties:

1. Its result has the same shape as its argument(s).
2. Each element of the result is defined by applying the function to the corresponding element of the argument.

5.5.1.1 Mathematical Elemental Functions

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

Nap Function	Formula	Description
`abs(`x`)`	$\lvert x \rvert$	Absolute value of x
`acos(`x`)`	$\arccos x$	Arc cosine of x, in the range 0 to π
`asin(`x`)`	$\arcsin x$	Arc sine of x, in the range -$\pi/2$ to $\pi/2$
`atan(`x`)`	$\arctan x$	Arc tangent of x, in the range -$\pi/2$ to $\pi/2$
`atan(`y`,`x`)`	$\arctan(y/x)$	Arc tangent of y/x, in the range $-\pi$ to π
`atan2(`y`,`x`)`	$\arctan(y/x)$	Alias for `atan`
`ceil(`x`)`	$\lceil x \rceil$	Smallest integer not less than x
`cos(`x`)`	$\cos x$	Cosine of x (x in radians)
`cosh(`x`)`	$\cosh x$	Hyperbolic cosine of x
`exp(`x`)`	e^x	e is base of natural logarithms
`floor(`x`)`	$\lfloor x \rfloor$	Largest integer not greater than x
`fmod(`x`,`y`)`	$x \bmod y$	Remainder after dividing x by y
`hypot(`x`,`y`)`	$\sqrt{x^2 + y^2}$	Length of hypotenuse
`isnan(`x`)`		1 if x is NaN, 0 otherwise
`log(`x`)`	$\log_e x$	Natural (base e) logarithm
`log(`x`,`y`)`	$\log_y x$	Logarithm with base y
`log10(`x`)`	$\log_{10} x$	Common (base 10) logarithm
`nint(`x`)`		Nearest integer to x
`pow(`x`,`y`)`	x^y	x raised to power y
`random(`x`)`		Random number r such that `0` $\leq r < x$
`round(`x`)`		Alias for `nint`
`sign(`x`)`	$(x > 0) - (x < 0)$	-1 if $x < 0$, 0 if $x = 0$, 1 if $x > 0$
`sin(`x`)`	$\sin x$	Sine of x (x in radians)
`sinh(`x`)`	$\sinh x$	Hyperbolic sine of x
`sqrt(`x`)`	$\sqrt{x}$	Square root
`tan(`x`)`	$\tan x$	Tangent of x (x in radians)
`tanh(`x`)`	$\tanh x$	Hyperbolic tangent of x

5.5.1.2 Data-type Conversion Functions

The data-type conversion functions are also elemental. These are `c8(`x`)`, `f32(`x`)`, `f64(`x`)`, `i8(`x`)`, `i16(`x`)`, `i32(`x`)`, `u8(`x`)`, `u16(`x`)` and `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
```

5.5.2 Reduction Functions

A *reduction* or *insert* function is one which has the effect of inserting a binary operator between the *cells* of its argument or sub-arrays of it. Such functions are termed *reductions* because the result has a rank which is one less than the argument.

The *cells* of an array of rank r are the sub-arrays of rank $r-1$. Thus the cells of a vector are its elements, while the cells of a matrix are its rows.

The Nap reduction functions are listed in the following table:

Function	Operator	Result
`max(`x`[ , `r`])`	`>>>`	Maximum of rank-r sub-arrays of x
`min(`x`[ , `r`])`	`<<<`	Minimum of rank-r sub-arrays of x
`prod(`x`[ , `r`])`	`*`	Product of rank-r sub-arrays of x
`sum(`x`[ , `r`])`	`+`	Sum of rank-r sub-arrays of x

The optional second argument r of reduction functions is called the *verb-rank* (as in J). It is used to specify the rank of the sub-arrays of x (if its rank exceeds 1) to which the reduction process is to be applied. If r is not specified then it defaults to the rank of x and the reduction is applied just once to x as a whole. For example, if x is a matrix then `sum(`x`)` gives the sum of each column, while `sum(`x`,1)` gives the sum of each row.

If the argument is a vector then its elements are the cells and the result is a scalar. For example if x is a vector then the operator '+' is inserted between its elements giving the scalar result

$$\texttt{sum(}x\texttt{)} = x_0 + x_1 + x_2 + \ldots = \sum x$$

The following shows the application of each of the four functions to a vector:

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

Consider the application of function `sum()` to a matrix (rank 2) argument x. The result is a vector (rank 1).

If r is not specified then the reduction is applied once to the entire argument. The cells of a matrix are its rows, so the result is

$$row_0 + row_1 + row_2 + \ldots$$

which gives the sum of each column.

If $r = 1$ then the reduction is applied to each rank-1 sub-array (row) of x and the results are concatenated giving

$$\sum row_0 \ \sum row_1 \ \sum row_2 \ \ldots$$

which gives the sum of each row.

This is demonstrated by:

```
% nap "mat = {
{2 5 0}
{6 7 1}
}"
::NAP::49-49
% [nap "sum mat"]
8 12 1
% [nap "sum(mat,1)"]
7 14
% [nap "sum(sum(mat))"]
21
```

The following example produces sums of a *3-dimensional* array:

```
% [nap "a3d = reshape(0 .. 23, {2 3 4})"]
 0  1  2  3
 4  5  6  7
 8  9 10 11

12 13 14 15
16 17 18 19
20 21 22 23
% [nap "sum(a3d)"]
12 14 16 18
20 22 24 26
28 30 32 34
% [nap "sum(a3d,1)"]
 6 22 38
54 70 86
% [nap "sum(a3d,2)"]
12 15 18 21
48 51 54 57
```

The following example shows how to test for '*any true*' and '*all true*'.

```
% [nap "bool = {{0 0 1}{0 1 1}{0 0 1}}"]
0 0 1
0 1 1
0 0 1
% [nap "prod(bool) == 1"]; # column all 1s?
0 0 1
% [nap "sum(bool) > 0"]; # column contains any 1?
0 1 1
```

Missing values in an argument are ignored by the reduction functions and thus do not produce missing values in the result. Note that the corresponding binary operations ('`>>>`', '`<<<`', '`*`' and '`+`') *do* produce a missing value if either operand is missing. The following example applies the four reduction functions to a vector with a missing value:

```
% [nap "max({-3 _ 5 1})"]
5
% [nap "min({-3 _ 5 1})"]
```

```
-3
% [nap "prod({-3 _ 5 1})"]
-15
% [nap "sum({-3 _ 5 1})"]
3
```

If there are no non-missing elements then the result is the *identity element* of the operation. The following shows this for each of the four reduction functions:

```
% [nap "max(f64{_})"]
-Inf
% [nap "min(f64{_})"]
Inf
% [nap "prod(f64{_})"]
1
% [nap "sum(f64{_})"]
0
```

5.5.3 Function `count(`x`[ , `r`])`

Function `count(`x`[ , `r`])` gives the number of non-missing elements in rank-r sub-arrays of x. It is similar to the reduction functions in many respects.

However unlike them, it *is* useful to specify the verb-rank $r = 0$. This means that each (rank 0) element is processed separately. This gives 1 if the element is present and 0 if it is missing. The result has the same shape as the argument. Let's name this 'is-present' mask 'p', so

$$p = \texttt{count(}x\texttt{,0)}$$

The result of `count(`x`)` (r absent) is defined as `sum(`p`)`.

If $r > 0$ then `count(`p`,`r`)` is defined as `sum(`p`,`r`)`.

An example with a vector argument is:

```
% nap "vec = {4 _ 2 -9 _ 6}"
::NAP::51-14
% [nap "count(vec, 0)"]
1 0 1 1 0 1
% [nap "count vec"]
4
```

An example with a matrix argument is:

```
% [nap "mat = {{0 _ _ 4}{_ _ 5 2}}"]
0 _ _ 4
_ _ 5 2
% [nap "count(mat, 0)"]
1 0 0 1
0 0 1 1
% [nap "count mat"]
1 0 1 2
% [nap "count(mat, 1)"]
2 2
```

Note that many simple statistics can be calculated using the reduction functions and `count()`. For example we can calculate arithmetic means of the above vector and matrix as follows:

```
% [nap "sum(vec) / count(vec)"]
0.75
% [nap "sum(mat) / count(mat)"]; # mean of each column
0 _ 5 3
% [nap "sum(mat,1) / count(mat,1)"]; # mean of each row
2 3.5
```

5.5.4 Scan Functions

A *scan* function produces a result with the same shape as its argument. Each element of the result is a reduction over part of the argument.

Nap has two scan functions `psum(`x`)` and `psum1(`x`[ , `r`])`. Both produce a result consisting of partial-sums. `psum(`x `)` produces *multi-directional* partial-sums of x. `psum1(`x`[ , `r`])` produces *uni-directional* partial-sums of rank-r sub-arrays of x.

If x is a vector then these two functions give the same result. Let r be this result. Each element of r is defined by

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

For example:

```
% nap "x = {2 7 1 3 8 2 5 0 2 5}"
::NAP::14-14
% [nap "psum(x)"] value
2 9 10 13 21 23 28 28 30 35
% [nap "psum1(x)"] value
2 9 10 13 21 23 28 28 30 35
```

Missing values are treated as zeros. E.g.

```
% [nap "psum{5 -9 _ 6 4}"]
5 -4 -4 2 6
```

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

```
% nap "ps = 0 // psum(x)"
::NAP::25-25
% $ps 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
```

Missing values can be handled as follows:

```
% nap "x = f64{2 7 1 _ 3 8 2 5 _ 0 2 5}"
::NAP::187-187
% nap "ps = 0 // psum(x)"
::NAP::192-192
% $ps value
0 2 9 10 10 13 21 23 28 28 28 30 35
% nap "psc = 0 // psum(isPresent x)"; # psum of counts
::NAP::203-203
% $psc value
0 1 2 3 3 4 5 6 7 7 8 9 10
% nap "i = 0 .. 9"
::NAP::209-209
% [nap "(ps(i+3) - ps(i)) / (psc(i+3) - psc(i))"] value
3.33333 4 2 5.5 4.33333 5 3.5 2.5 1 2.33333
```

The library function `moving-average` (see section 10.3.3) generalises this moving-average technique to any rank.

If the rank of x exceeds 1 then `psum(`x`)` and `psum1(`x`[,`r`])` give different results. Function `psum1` sums in a single direction defined by the verb-rank in the same manner as the reduction functions. If x is a matrix and r is the result `psum(`x`)`, then each element of r is defined by

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

The following example uses the matrix `mat` defined above:

```
% $mat
2 5 0
6 7 1
% [nap "psum1(mat)"]
 2  5  0
 8 12  1
% [nap "psum1(mat, 1)"]
2 7 7
6 7 1
% [nap "psum(mat)"]
 2  7  7
 8 20 21
```

Other similar *scan* functions can be defined for partial products and so on. However Nap currently has only `psum` and `psum1`.

5.5.5 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(`x`))`
`rank(`x`)`	Number of dimensions = `nels(shape(`x`))`
`shape(`x`)`	Vector of dimension sizes

5.5.6 Functions which change shape or order

Function	Result
`sort(`x[`,`r]`)`	Sort rank-r sub-arrays of x into ascending order over most significant dimension
`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 "sort({{3 0 2}{1 9 1}})"]; # sort each column
1 0 1
3 9 2
% [nap "sort({{3 0 2}{1 9 1}}, 1)"]; # sort each row
0 2 3
1 1 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
% [nap "a3d = reshape(0 .. 23, {2 3 4})"]
 0  1  2  3
 4  5  6  7
 8  9 10 11

12 13 14 15
16 17 18 19
20 21 22 23
% [nap "transpose a3d"]
 0 12
 4 16
 8 20

 1 13
 5 17
 9 21

 2 14
 6 18
10 22

 3 15
 7 19
11 23
% [nap "transpose(a3d, {0 2 1})"]
 0  4  8
 1  5  9
 2  6 10
 3  7 11

12 16 20
13 17 21
14 18 22
15 19 23
```

5.5.7 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.

Let's solve the following system of two linear equations:

$$\begin{aligned} 3x - 4y &= 20 \\ -5x + 8y &= -36 \end{aligned}$$

This can be done as follows:

```
% 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
```

5.5.8 Correlation

The functions `correlation` and `moving_correlation` both calculate Pearson product-moment correlations. Function `correlation` calculates correlations between variables defined by the dimensions of its (one or two) arguments. Function `moving_correlation` calculates pattern (spacial) correlations between a (vector or matrix) window variable and variables defined by moving a window of the same shape around a larger array of the same rank.

Both functions handle missing values by omitting cases where one or both values are missing. The result consists of two layers. Layer 0 contains the correlation values themselves. Layer 1 contains the corresponding number of (non-missing) cases (sample size n) used to calculate these values.

Both functions produce an `f64` result if any of the data is `f64`, but otherwise the result is `f32`.

5.5.8.1 Function `correlation`(x[,y])

If y is not specified then it defaults to x.

The 1st (most significant) dimensions of x and y must have the same size, since this corresponds to the number of cases. (For time-series this dimension is time.) The remaining dimensions (if any) of x and y are essentially merged into column dimensions, but do appear in the result.

For example, let x be an 80×3 matrix and y a 80×5 matrix. The command

```
nap "r = correlation(x, y)"
```

produces a $2 \times 3 \times 5$ array r. r_{0ij} is the correlation between column i of x and column j of y. r_{1ij} is the number of cases (n) used to calculate r_{0ij}.

A simple example is:

```
% [nap "correlation({1 3 _ 6 6}, {6 6 4 2 3})"]
-0.924138 4
```

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.

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.

5.5.8.2 Function `moving_correlation`(x,y,[lag_0[,lag_1]])

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.

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)

5.5.9 Geometry

5.5.9.1 Testing whether points are in polygon

Function `inPolygon`(x, y, p) tests whether the points defined by x and y are inside the polygon defined by p. The result is

- -1 if (x, y) is outside the polygon.
- 0 if (x, y) is exactly on an edge (boundary) of the polygon.
- 1 if (x, y) is inside the polygon.

The algorithm is an extension of W. Randolph Franklin's PNPOLY. `PNPOLY` classifies points into only two categories, *inside* and *outside*. Points exactly on an edge can be classified either way. The modified algorithm in `inPolygon` does detect edge points.

The ranks of x and y can differ provided their trailing dimensions match. The shape of the result is that of the one of higher rank. If an element of x or y is missing then the corresponding element of the result is missing.

The argument p is an $n \times m$ matrix defining the n vertices (x, y) of the polygon. There must be at least one vertex. The number of columns (m) must be at least 2. Column 0 contains x. Column 1 contains y. Any other columns are ignored.

The following example tests whether the points (1,2), (2,2), (3,2), (4,2), (5,2), (6,2) are in the triangle with vertices (0,0), (5,0), (5,5):

```
% [nap "inPolygon(1 .. 6, 2, {{0 0}{5 0}{5 5}})"]
-1 0 1 1 0 -1
```

5.5.9.2 Triangulation

Triangulation is the process of joining scattered (x, y) points (called *sites*) to form triangles. One important use of triangulation is interpolation of (x, y, z) data. Function `scattered2grid` (section 10.5.20) interpolates by defining a plane for each triangle produced by triangulation.

The Delaunay triangulation is the optimal triangulation in various senses. For example, it maximises the minimum angle. Delaunay triangulation is closely related to Voronoi Diagrams which consist of polygons around each site. The points within each polygon are those which are closer to this site than to any other site. A good survey of Delaunay triangulation and Voronoi diagrams is given by

> Franz Aurenhammer, *Voronoi diagrams – a survey of a fundamental geometric data structure*, ACM Computing Surveys, Volume 23, Issue 3 (September 1991), pp345-405, 1991

Function `triangulate` gives the triangles defined by Delaunay triangulation.

Function `triangulate_edges` gives the edges defined by the same Delaunay triangulation.

The code of both functions is based on a program written by Geoff Leach, Department of Computer Science, RMIT, Melbourne, Australia. This program implements a very efficient Delaunay triangulation algorithm that is $O(n \log n)$ time and $O(n)$ space. This worst-case optimal divide-and-conquer algorithm is described in

> Guibas, L. and Stolfi, J., *Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams* , Proceedings of the fifteenth annual ACM symposium on theory of computing, pp221-234, 1983.

Function	**Result**
`triangulate(`*sites*`)`	$t \times 3$ matrix of site indices defining t triangles
`triangulate_edges(`*sites*`)`	$e \times 2$ matrix of site indices defining e edges

The argument *sites* is an $n \times m$ matrix defining the n points (x,y). The number of columns (m) must be at least 2. Column 0 contains x. Column 1 contains y. Any other columns are ignored. The row number is called the *site index* and ranges from 0 to n-1.

The following example triangulates the sites (0,0), (2,1), (0,1) and (1,2). These have site indices 0, 1, 2 and 3 respectively. Function `triangulate` gives the site indices of the vertices of each of two triangles. Function `triangulate_edges` gives the site indices of the vertices of each of the five edges of these same two triangles.

```
% [nap "triangulate({{0 0}{2 1}{0 1}{1 2}})"]
0 1 2
1 2 3
% [nap "triangulate_edges({{0 0}{2 1}{0 1}{1 2}})"]
0 2
0 1
1 3
1 2
2 3
```

5.5.10 Grid Functions

There are currently just two closely related grid functions, `invert_grid` and `invert_grid_no_trim`. These can be applied to

- one-dimensional data to define a piecewise-linear mapping as the inverse of a given piecewise-linear mapping.
- two-dimensional data to define a piecewise-bilinear mapping as the inverse of a given piecewise-bilinear mapping.

There is only a minor difference between these two functions. Function `invert_grid` suppresses edges (rows and columns in 2D case) containing nothing except missing values. Function `invert_grid_no_trim` does no such trimming.

In the 1D case we have a piecewise-linear mapping from x to y, and we want a piecewise-linear mapping from y to x. The functions are called by
`invert_grid(`y`,`$ycv`)`
or
`invert_grid_no_trim(`y`,`$ycv`)`
where y is the known mapping (and has a coordinate variable corresponding to x)
and ycv is the desired new y coordinate variable.

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 $(5, 4)$. The difference between `invert_grid` and `invert_grid_no_trim` is shown by attempting to extrapolate to $y = -1$ with both these functions. Both produce the inverse mapping from y to x defined by the four lines joining the five points (0,0), (2,1), (3,2), (4,3) and (5,4). Function `invert_grid_no_trim` generates the requested point for $y = -1$ (with x missing) whereas `invert_grid` suppresses this point.

```
% nap "y = {0 1 4}"
::NAP::56-56
% $y set coo "{0 2 5}"
% [nap "coordinate_variable(y) /// y"]
0 2 5
0 1 4
% [nap "ycv = -1 .. 4"] value
-1 0 1 2 3 4
% nap "x = invert_grid(y,ycv)"
::NAP::75-75
% $x all
::NAP::75-75  f32  MissingValue: NaN  References: 1  Unit: (NULL)
Link: ::NAP::76-76
Dimension 0   Size: 5      Name: (NULL)    Coordinate-variable: ::NAP::72-72
Value:
0 2 3 4 5
% [nap "coordinate_variable(x) /// x"]
0 1 2 3 4
0 2 3 4 5
% nap "xnt = invert_grid_no_trim(y,ycv)"
::NAP::86-86
% $xnt all
::NAP::86-86  f32  MissingValue: NaN  References: 1  Unit: (NULL)
Link: ::NAP::87-87
Dimension 0   Size: 6      Name: (NULL)    Coordinate-variable: ::NAP::83-83
Value:
_ 0 2 3 4 5
% [nap "coordinate_variable(xnt) /// xnt"]
-1  0  1  2  3  4
 _  0  2  3  4  5
```

In the 2D case, the functions are called by
`invert_grid(`y`,`ycv`,`x`,`xcv`)`,
or
`invert_grid_no_trim(`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 has the specified coordinate-variable ycv
- Dimension 1 has the specified coordinate-variable xcv.
- Dimension 2 is of size 2, corresponding to the i and j mappings. We can think of the result as two matrices defining mappings from xy space to i and j respectively.

The following 2D example shows how a satellite image can be mapped to latitude/longitude space (i.e. a *Cylindrical Equidistant* map projection). Note that the terms *line* and *pixel* refer to the row and column of a raw satellite image respectively. We are interested in an input region bounded by line 40, line 50, pixel 1 and pixel 21. Assume we know the latitude and longitude on a 3×3 grid corresponding to lines 40, 50, 60 and pixels 1, 11, 21. The following defines and displays an inverse grid for latitudes 40°S, 30°S, 25°S and longitudes 150°E, 160°E, 170°E, 180°E:

```
nap "line = {40 50 60}"
nap "pixel = {1 11 21}"
nap "lat_grid = {{-40 -50 _}{-30 -40 -50}{-20 -30 -40}}"
$lat_grid set coo line pixel
nap "lon_grid = {{200 180 _}{180 160 140}{160 140 120}}"
$lon_grid set coo line pixel
nap "lat_cv = {-40 -30 -25}"
$lat_cv set unit degrees_north
nap "lon_cv = 150 .. 180 ... 10"
$lon_cv set unit degrees_east
nap "ig = invert_grid(lat_grid, lat_cv, lon_grid, lon_cv)"
$ig all
```

This displays the following inverse grid:

```
::NAP::46-46  f32  MissingValue: NaN  References: 1  Unit: (NULL)
Link: ::NAP::47-47
Dimension 0   Size: 3      Name: (NULL)    Coordinate-variable: ::NAP::36-36
Dimension 1   Size: 4      Name: (NULL)    Coordinate-variable: ::NAP::39-39
Dimension 2   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Value:
52.5 13.5
50.0 11.0
47.5  8.5
45.0  6.0

57.5  8.5
55.0  6.0
52.5  3.5
50.0  1.0

60.0  6.0
57.5  3.5
55.0  1.0
   -    -
```

The following displays the coordinate variables of this inverse grid `ig`:

```
% [$ig coo 0] all
::NAP::36-36  f32  MissingValue: NaN  References: 1  Unit: degrees_north
```

```
Dimension 0   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
-40 -30 -25
% [$ig coo 1] all
::NAP::39-39  f32  MissingValue: NaN  References: 1  Unit: degrees_east
Dimension 0   Size: 4      Name: (NULL)    Coordinate-variable: (NULL)
Value:
150 160 170 180
```

The first row of `ig` is

```
52.5 13.5
```

Let us use these values as indirect indices of the original latitude and longitude grids:

```
% [nap "lat_grid(@52.5, @13.5)"]
-40
% [nap "lon_grid(@52.5, @13.5)"]
150
```

Note that these results correspond to the initial values of the coordinate variables of `ig`.
Now let us define a small extract from a raw satellite image:

```
nap "raw = {
{99 91 91 90}
{95 95 92 91}
{90 89 88 88}
}"
nap "line = 50 .. 52"
nap "pixel = 1 .. 4"
$raw set coo line pixel
```

The latitudes and longitudes at these points are given by

```
% [nap "lat_grid(@line, @pixel)"]
-30 -31 -32 -33
-29 -30 -31 -32
-28 -29 -30 -31
% [nap "lon_grid(@line, @pixel)"]
180 178 176 174
178 176 174 172
176 174 172 170
```

We can map this image to a *Cylindrical Equidistant* projection as follows:

```
% nap "latitude = f32(-29 .. -33)"
::NAP::135-135
% nap "longitude = f32(170 .. 180)"
::NAP::142-142
% nap "cyl_eq = raw(@ig(@latitude, @longitude, ))"
::NAP::166-166
% $cyl_eq all -f %.1f -col 11
::NAP::166-166  f32  MissingValue: NaN  References: 1  Unit: (NULL)
Dimension 0   Size: 5      Name: (NULL)    Coordinate-variable: ::NAP::152-152
Dimension 1   Size: 11     Name: (NULL)    Coordinate-variable: ::NAP::153-153
Value:
91.0    _    _    _    _    _    _    _    _    _    _
89.2 88.7 88.0 89.4 91.0 92.9 95.0 94.5 95.0 96.5 99.0
88.0 88.8 89.8 90.8 92.0 92.3 92.2 91.8 91.0 92.1 92.2
91.0 91.1 91.0 91.0 91.0 91.0 91.0 90.3 89.8 89.3 89.0
```

```
95.0 94.7 93.8 92.2 90.0 89.7 89.2 88.7 88.0 89.4 91.0
% [$cyl_eq coo 0] all
::NAP::152-152  f32  MissingValue: NaN  References: 1  Unit: degrees_north
Dimension 0   Size: 5      Name: (NULL)    Coordinate-variable: (NULL)
Value:
-29 -30 -31 -32 -33
% [$cyl_eq coo 1] all -col 11
::NAP::153-153  f32  MissingValue: NaN  References: 1  Unit: degrees_east
Dimension 0   Size: 11     Name: (NULL)    Coordinate-variable: (NULL)
Value:
170 171 172 173 174 175 176 177 178 179 180
```

5.5.11 Cartographic Projection Functions

There are two built-in cartographic (map) projection functions corresponding to forward and inverse projections:

- `cart_proj_fwd`(*proj_spec*, *lat*, *lon*)
- `cart_proj_inv`(*proj_spec*, *x*, *y*)

These are based on the PROJ.4 Cartographic Projections library originally written by Gerald Evenden then of the USGS. This software supports over seventy projections and various parameters. The primary PROJ.4 home page is http://www.remotesensing.org/proj. There is a mirror at http://proj.maptools.org. Note the links to documentation, which is essential to use these Nap functions.

5.5.11.1 Function `cart_proj_fwd`(*proj_spec*, *lat*, *lon*)

proj_spec: `c8` NAO containing a PROJ.4 projection specification.
lat: latitudes.
lon: longitudes.

lon and *lat* can have any ranks and these can differ. The sizes of their trailing dimensions must match. If *lon* or *lat* has no unit then this defaults to degrees.

The result contains eastings and northings in metres. It's rank is 1+(`rank`(*lat*) `>>>` `rank`(*lon*)). The new least significant dimension has size 2. Eastings are in column 0. Northings are in column 1.

The following example converts latitude 10.5°S, longitude 97°E, to the Mercator Projection. The resulting easting is 10797990.61. The resulting northing is −1167671.00.

```
% [nap "cart_proj_fwd('proj=merc', -10.5, 97)"] -f %.2f
10797990.61 -1167671.00
```

The following example converts six points to the Universal Transverse Mercator (UTM) projection for zone 55 in the Southern Hemisphere. Note that the latitude argument has rank 1, while the longitude argument has rank 2.

```
% nap "spec = 'proj=utm south zone=55'"
::NAP::141-141
% nap "latitude = {-39 -38 -35}"
::NAP::143-143
% [nap "longitude = {{149 145 144}{143 148 142}}"]
149 145 144
143 148 142
% [nap "xy = cart_proj_fwd(spec, latitude, longitude)"] -f %.3f
 673190.900 5681320.708
 324396.629 5792297.633
 226201.904 6122843.308

 153573.959 5675606.444
```

```
 587798.418 5793713.242
  43542.271 6115516.008
```

5.5.11.2 Function `cart_proj_inv`(*proj_spec*, x, y)

proj_spec: `c8` NAO containing a PROJ.4 projection specification.
x: eastings (metres).
y: northings (metres).

x and y can have any ranks and these can differ. The sizes of their trailing dimensions must match.

The result contains latitudes and longitudes in degrees. It's rank is $1 + (\texttt{rank}(x)$ `>>>` $\texttt{rank}(y))$. The new least significant dimension has size 2. Latitudes are in column 0. Longitudes are in column 1.

The following example is the inverse of the first example in section 5.5.11.1.

```
% [nap "cart_proj_inv('proj=merc', 10797990.61, -1167671)"]
-10.5 97
```

The following example is the inverse of the second example in section 5.5.11.1.

```
% [nap "cart_proj_inv(spec, xy(,,0), xy(,,1))"]
-39 149
-38 145
-35 144

-39 143
-38 148
-35 142
```

5.5.12 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

5.5.12.1 Function `open_box`(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
```

5.5.12.2 Functions `prune(`x`)` and `pad(`x`)`

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
```

5.5.13 Morphological Functions

The basic concepts are explained in the MATLAB documentation on Morphological Operations at http://www.mathworks.com.

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

5.5.13.1 Morphological Binary Dilation and Erosion

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

5.5.13.2 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.

Chapter 6

Object-Oriented Commands (OOCs)

6.1 Introduction

Object-Oriented Commands (OOCs) are used to

- display the data in a NAO
- display other information (metadata e.g. data-type, dimensions) about a NAO
- change data or other aspects of a NAO
- write data from a NAO to a file, etc.

6.2 OOC Methods which return Data Values (with or without metadata)

6.2.1 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: Use internal format if any, else `""` meaning automatic)
`-columns` *int*: maximum number of columns (default: 6) (`-1`: no limit)
`-lines` *int*: maximum number of 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*

6.2.1.1 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
```

6.2.2 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: Use internal format if any, else `""` meaning automatic)
`-columns` *int*: maximum number of columns (default: `-1` i.e. no limit)
`-lines` *int*: maximum number of 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

6.2.2.1 Example

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

6.2.3 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: Use internal format if any, else `""` meaning automatic)
`-columns` *int*: maximum number of columns (default: `6`) (`-1`: no limit)
`-lines` *int*: maximum number of 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

6.2.3.1 Examples

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 }
}
```

6.2.4 Format Conversion Strings

The `-format` option (or NAO internal *format* field) 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* $\geq$ *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
```

6.3 OOC Methods which return Metadata

6.3.1 Introduction

The following code defines vectors '`x`' and '`y`' for use in the examples below:

```
% 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
```

6.3.2 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 (continued from above):

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

6.3.3 Method `count`

ooc_name `count` ?`-keep`?

This returns the reference count.

Example (continued from above):

```
% $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`

6.3.4 Method `datatype`

ooc_name `datatype`
This returns the data-type.
Example (continued from above):

```
% $x dat
f64
```

6.3.5 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 (continued from above):

```
% $y dim
x
```

6.3.6 Method `format`

ooc_name `format`
This returns the C format for printing the NAO.
Example (continued from above):

```
% $y set format %.4f
% $y format
%.4f
% $y value
0.0000 0.2500 1.0000 2.2500 4.0000
```

6.3.7 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 (continued from above):

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

6.3.8 Method `label`

ooc_name `label`
This returns the label (title, etc.) of the NAO.
Example (continued from above):

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

6.3.9 Method `link`

ooc_name `link`
This returns the OOC-name of the link NAO.
Example (continued from above):

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

6.3.10 Method `missing`

ooc_name `missing`
This returns the missing value. This is the value used to indicate null or undefined data.
Example (continued from above):

```
% $y miss
NaN
```

6.3.11 Method `ooc`

ooc_name `ooc` ?`-keep`?
This returns the OOC-name of the NAO.
Example (continued from above):

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

6.3.12 Method `rank`

ooc_name `rank`

This returns the rank (number of dimensions).
Example (continued from above):

```
% $y rank
1
```

6.3.13 Method `sequence`

ooc_name `sequence` ?`-keep`?
This returns the sequence number of the NAO. E.g. 42 for `::NAP::42-9`
Example (continued from above):

```
% $y seq
61
```

6.3.14 Method `shape`

ooc_name `shape`

This returns the shape, which is a vector of dimension sizes.
Example (continued from above):

```
% $y shape
5
```

6.3.15 Method `slot`

ooc_name `slot` ?`-keep`?

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

Example (continued from above):

```
% $y sl
61
```

6.3.16 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) i.e. *Arithmetic Progression*

Example (continued from above):

```
% $x step
AP
% $y step
>= 0
```

6.3.17 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. At the moment Nap makes only limited use of the unit attribute. For example coordinate variables with the unit `degrees_east` are recognised at longitudes and treated differently.

Example (continued from above):

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

6.4 OOC Methods which Modify NAO

6.4.1 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 # 8)"] draw "{{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 0 1 0 0 0
0 0 1 0 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
```

6.4.2 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
```

6.4.3 Method `set`

ooc_name `set` *attribute arg arg* ...

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

6.4.3.1 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 set coo

An example is provided in section 6.3.2.

6.4.3.2 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 (even if new value of count is 0 or less)
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')

6.4.3.3 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 set dim
Example (continued from section 6.3):

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

6.4.3.4 set format

ooc_name `set format` ?*string*?
This sets the C format for printing. The default is NULL i.e. no format.
An example is provided in section 6.3.6.

6.4.3.5 set label

ooc_name `set label` ?*string*?
This sets the label (title). The default is NULL i.e. no label.
An example is provided in section 6.3.8.

6.4.3.6 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.
An example is provided in section 6.3.9.

6.4.3.7 Set missing

ooc_name `set missing` ?*value*?
This sets the missing value. The default is NULL i.e. no missing value.
Example (continued from section 6.3):

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

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

6.4.3.8 Set unit

ooc_name `set unit` ?*unit*?
This sets the unit of measure. The default is NULL i.e. no unit.
An example is provided in section 6.3.17.

6.4.3.9 Set value

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

This sets the value of data elements selected by NAO *index* to new values copied from successive elements of NAO *value*.

If *value* is absent or `""` it is treated as the null (missing) value. Elements of *value* are recycled if necessary.

If *index* is absent or `""` it is treated as the whole array. The *index* expression can include the operators '`,`' (giving cross-product-indexing as discussed in section 5.4.2.4) and '`@@`' (giving indirect indexing as discussed in section 5.4.3). However full-indexing (see section 5.4.2.3) is not yet implemented.

Note that the Nap assignment operator '`=`' does not allow its left operand to be indexed, as in '`v(2) = 9`'. However the '`set value`' method does provide a way to modify all or part of an array. The following example sets element 2 of `v` to 9:

```
% nap "v = {3 6 4 7}"
::NAP::52-52
% $v set value 9 2
% $v
3 6 9 7
```

The following example changes elements 1, 2 and 3. Note how the elements of *value* are recycled.

```
% $v set value "{8 5}" "1 .. 3"
% $v
3 8 5 8
```

The following example illustrates indirect indexing:

```
% $v set coo "10 .. 40 ... 10"
% $v set value 0 "@@{20 40}"
% $v
3 0 5 0
```

The following example illustrates cross-product indexing:

```
% [nap "matrix = {{0 2 4}{1 3 5}}"]
0 2 4
1 3 5
% $matrix set v 9 0,2; # Note that 'value' can be abbreviated to 'v'
% $matrix
0 2 9
1 3 5
% $matrix set v "-1 .. -9" ",{0 2}"
% $matrix
-1  2 -2
-3  3 -4
```

6.5 OOC Methods which Write to File

The examples in this section use the NAOs created by:

```
% nap "x = 0 .. 2 ... 0.5"
::NAP::21-21
% nap "y = x ** 2"
::NAP::24-24
% $y set coord x
% $y set unit m^2
% $y set label "areas of squares"
```

```
% $y all
::NAP::24-24  f64  MissingValue: NaN  References: 1  Unit: m^2
areas of squares
Dimension 0   Size: 5      Name: x         Coordinate-variable: ::NAP::21-21
Value:
0 0.25 1 2.25 4
```

6.5.1 Method `binary`

ooc_name `binary` ?*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 binary $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
```

6.5.2 Method `hdf`

ooc_name `hdf` ?*switches*? *filename name*

This writes data from the NAO to an SDS or attribute within an HDF file named *filename*. The HDF file format is discussed in section 2.5.

name is the name of an SDS or attribute and has the form:

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

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` *expr*: HDF `valid_range`
`-scale` *expr*: HDF `scale_factor`
`-offset` *expr*: 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 specified then *expr* must be an index compatible with the rank r of the SDS. If $r = 1$ then *expr* must be a *shape-preserving* index, as described in section 5.4.2.1. If $r > 1$ then *expr* must be a *cross-product* index (as described in section 5.4.2.4) containing r scalar and vector elements. The vector elements correspond to dimensions which exist in both the SDS and the NAO. The scalar elements correspond to dimensions which exist in the SDS but not the NAO.

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.

If '`-scale` *expr*' and/or '`-offset` *expr*' are specified then these values are:

- written as attributes `scale_factor` and `add_offset`
- used to scale the data before writing it.

Example:

```
% $y hdf y.hdf y
% exec hdp dumpsds y.hdf
File name: y.hdf

Variable Name = y
         Index = 0
         Type= 64-bit floating point
         Ref. = 2
         Rank = 1
         Number of attributes = 3
         Dim0: Name=x
                 Size = 5
                 Scale Type = 64-bit floating point
                 Number of attributes = 0
         Attr0: Name = _FillValue
                 Type = 64-bit floating point
                 Count= 1
                 Value = DoubleInf
         Attr1: Name = long_name
                 Type = 8-bit signed char
                 Count= 16
                 Value = areas of squares
         Attr2: Name = units
                 Type = 8-bit signed char
                 Count= 3
                 Value = m^2
         Data :
                0.000000 0.250000 1.000000 2.250000 4.000000


Dimension Variable Name = x
         Index = 1
         Scale Type= 64-bit floating point
         Ref. = 3
         Rank = 1
         Number of attributes = 0
         Dim0: Name=x
                 Size = 5
         Data :
                0.000000 0.500000 1.000000 1.500000 2.000000
```

Procedure `put16` (see section 11.3.5.1) can be used to write an automatically-scaled 16-bit SDS to an HDF file.

6.5.3 Method `netcdf`

ooc_name `netcdf` ?*switches*? *filename name*

This writes data from the NAO to a variable or attribute within a netCDF file named *filename*. The netCDF file format is discussed in section 2.5.

name is the name of a netCDF variable or attribute and has the form:

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

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` *expr*: netCDF `valid_range`
`-scale` *expr*: netCDF `scale_factor`
`-offset` *expr*: 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 specified then *expr* must be an index compatible with the rank r of the netCDF variable. If $r = 1$ then *expr* must be a *shape-preserving* index, as described in section 5.4.2.1. If $r > 1$ then *expr* must be a *cross-product* index (as described in section 5.4.2.4) containing r scalar and vector elements. The vector elements correspond to dimensions which exist in both the netCDF variable and the NAO. The scalar elements correspond to dimensions which exist in the netCDF variable but not the NAO.

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.

If '`-scale` *expr*' and/or '`-offset` *expr*' are specified then these values are:

- written as attributes `scale_factor` and `add_offset`
- used to scale the data before writing it.

Example:

```
% $y netcdf y.nc y -coordinateVariable "x // {3 6 9}"
% exec ncdump y.nc
netcdf y {
dimensions:
        x = 8 ;
variables:
        double x(x) ;
        double y(x) ;
                y:_FillValue = nan ;
                y:long_name = "areas of squares" ;
                y:units = "m^2" ;
data:

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

 y = 0, 0.25, 1, 2.25, 4, _, _, _ ;
}
```

Note that the netCDF variables `x` and `y` are dimensioned to 8 (the shape of the argument specified by the `-coordinateVariable` option).

Procedure `put16` (see section 11.3.5.1) can be used to write an automatically-scaled 16-bit variable to a netCDF file.

6.5.4 Method `swap`

ooc_name `swap` ?*tcl_channel*?

Method `swap` is like method `binary`, except that bytes are swapped. This enables writing of data to be read on a machine with opposite byte-order within words.

Chapter 7

Reading Files using nap_get Command

7.1 Introduction

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`, `netcdf` or `swap`.

The HDF and netCDF file formats are discussed in section 2.5. The `netcdf` option supports (but only on the Linux Intel 386 platform at the time of writing) access to remote data provided by an OPeNDAP (a.k.a. DODS) server. In this case the *filename* is a URL rather than the pathname of a local file. There is an OPeNDAP example at the end of section 7.3 (*Reading netCDF Data*).

7.2 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 containing a single element, the file size.

The command
`nap_get swap` *channel* [*datatype* [*shape*]]
is similar, except that bytes are swapped. This enables reading of data written on a machine with opposite byte-order within words.

The following example first writes six 32-bit floating-point values to a file using the OOC `binary` method. It then reads them back into a NAO named '`in`' using '`nap_get binary`':

```
% set file [open float.dat w]
filee15170
% [nap "f32{1.5 -3 0 2 4 5}"] binary $file
% close $file
% set file [open float.dat]
filee15170
% nap "in = [nap_get binary $file f32]"
::NAP::22-22
% close $file
% $in all
::NAP::22-22  f32  MissingValue: NaN  References: 1
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
```

7.3 Reading netCDF Data

NetCDF data is read using the command
`nap_get netcdf` *filename name* [*index* [*raw*]]
filename is either:

- pathname of a local file
- URL of a remote OPeNDAP server

name is the name of a variable or attribute and has one of the following forms:

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

A single-element attribute gives a scalar. Other attributes give vectors. Neither *index* nor *raw* can be specified for attributes.

A variable with no *index* gives the entire variable. If *index* is specified then it selects using *cross-product* indexing, as described in section 5.4.2.4.

If *raw* is 1 then the result contains raw data read from the file. If *raw* is 0 (default) then this data is transformed using the attributes `scale_factor`, `add_offset`, `_FillValue`, `valid_min`, `valid_max` and `valid_range` if any of these are present.

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
```

Now let's specify 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
```

The following shows the different effects of a single-element vector index and a scalar index with the same value:

```
% [nap_get netcdf int.nc vec "{1}"] all
::NAP::65-65  i32  MissingValue: -2147483648  References: 0
Dimension 0   Size: 1      Name: n         Coordinate-variable: ::NAP::69-69
Value:
-9
% [nap_get netcdf int.nc vec "1"] all
::NAP::83-83  i32  MissingValue: -2147483648  References: 0
Value:
-9
```

The following is an OPeNDAP (a.k.a. DODS) example. It displays the altitude of the south pole.

```
% set url \
http://www.marine.csiro.au/dods/nph-dods/dods-data/climatology-netcdf/etopo5.nc
% [nap_get netcdf $url height "@@-90, 0"]
2810
```

7.4 Reading HDF Data

HDF data is read using the command
`nap_get hdf` *filename name* [*index* [*raw*]]
filename is the pathname of a local file.
name is the name of an SDS or attribute and has one of the following forms:

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

A single-element attribute gives a scalar. Other attributes give vectors. Neither *index* nor *raw* can be specified for attributes.

An SDS with no *index* gives the entire SDS. If *index* is specified then it selects using *cross-product* indexing, as described in section 5.4.2.4.

If *raw* is 1 then the result contains raw data read from the file. If *raw* is 0 (default) then this data is transformed using the attributes `scale_factor`, `add_offset`, `_FillValue`, `valid_min`, `valid_max` and `valid_range` if any of these are present.

The following example writes data to an HDF file using the OOC `hdf` method. Then `nap_get hdf` is used with various index values (including default) to read the data back into temporary NAOs (which are deleted after being displayed):

```
% [nap "f64{{1 0 9}{3 2 -1}}"] hdf mat.hdf mat64
% [nap_get hdf mat.hdf mat64] all; # default index giving whole SDS
::NAP::47-47  f64  MissingValue: NaN  References: 0
Dimension 0   Size: 2      Name: fakeDim0  Coordinate-variable: (NULL)
Dimension 1   Size: 3      Name: fakeDim1  Coordinate-variable: (NULL)
Value:
 1  0  9
 3  2 -1
% [nap_get hdf mat.hdf mat64 "1,0"] all; # select element
giving scalar
::NAP::78-78  f64  MissingValue: NaN  References: 0
Value:
3
```

```
% [nap_get hdf mat.hdf mat64 "{1},0"] all; # select element
giving vector
::NAP::102-102  f64  MissingValue: NaN  References: 0
Dimension 0   Size: 1      Name: fakeDim0  Coordinate-variable: ::NAP::92-92
Value:
3
% [nap_get hdf mat.hdf mat64 "0,"] all; # select row
::NAP::123-123  f64  MissingValue: NaN  References: 0
Dimension 0   Size: 3      Name: fakeDim1  Coordinate-variable: (NULL)
Value:
1 0 9
% [nap_get hdf mat.hdf mat64 ",2"] all; # select column
::NAP::147-147  f64  MissingValue: NaN  References: 0
Dimension 0   Size: 2      Name: fakeDim0  Coordinate-variable: (NULL)
Value:
9 -1
% [nap_get hdf mat.hdf mat64 ",{0 2}"] all; # select
sub-matrix
::NAP::154-154  f64  MissingValue: NaN  References: 0
Dimension 0   Size: 2      Name: fakeDim0  Coordinate-variable: (NULL)
Dimension 1   Size: 2      Name: fakeDim1  Coordinate-variable: ::NAP::161-161
Value:
 1  9
 3 -1
```

7.5 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/SDSs
`:`	attributes
`^:`	global attributes
`.:`	non-global attributes

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

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

7.6 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 SDS/variable 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 SDS/variable 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 SDS/variable 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 SDS/variable in the specified file.

The command
`nap_get hdf -coordinate` *filename sdsname dim_name|dim_number*
or
`nap_get netcdf -coordinate` *filename varname dim_name|dim_number*
returns the coordinate variable NAO corresponding to the specified dimension of the specified SDS/variable in the specified file.

Chapter 8

Other Nap Commands

8.1 The `nap_info` Command

The `nap_info` command provides information (mainly about the Nap system). There are three variants which are shown in the following example:

```
% nap_info bytes
2692 2920 68216 68224
% nap_info seconds
12.519
% nap_info sequence
15
```

The command '`nap_info bytes`' provides information about Nap's use of memory. This use falls into the following three categories:

- direct use for *NAOs* within a Tcl interpreter
- *other* purposes local to a Tcl interpreter
- *global* purposes

There can be multiple Tcl interpreters, each with its own Nap. No statistics are available on *global* memory usage. The command '`nap_info bytes`' returns the following four numbers relating to Nap's memory usage within the current Tcl interpreter:

1. current number of bytes of memory being used by *NAOs* (2692 in above example)
2. maximum number of bytes used by *NAOs* at any time so far (2920 in above example)
3. current number of bytes of memory being used for *other* purposes (68216 in above example)
4. maximum number of bytes used for *other* purposes at any time so far (68224 in above example)

The command '`nap_info seconds`' returns the processor time used by tcl, as defined by the C function '`clock()`'.

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

8.2 The `nap_land_flag` Command

The `nap_land_flag` command defines land/sea masks. The data has an accuracy of 0.01° of latitude/longitude.

This command is normally used via the functions `is_land` (see section 10.4.4.1.1) and `is_coast` (see section 10.4.4.1.2) rather than directly.

The `nap_land_flag` command is based on code which was originally written by Dr. Chris. Mutlow at the Rutherford Appleton Laboratory in England. This code has been adapted for Nap by Peter Turner and Harvey Davies.

Chapter 9

Defining New Nap Commands and Functions

9.1 Writing Procedures to be called as Commands or Functions

9.1.1 Introduction

One can write a Tcl procedure which defines a new Nap function or replaces a built-in Nap function. Of course it is also possible to write a Tcl procedure which is called in the normal Tcl manner (as a Tcl command) to do something related to Nap.

The following sections include various examples of procedure definition directly from the command-line. Of course in practice one would normally create such code in files, which would be sourced.

Tcl has facilities for automatically defining undefined commands when an attempt is made to execute them. In particular, the array `auto_index` contains the commands to define indexed commands.

Users with small libraries of their own procedures may prefer to simply source the relevant files as part of Tcl startup. The startup files distributed with Nap automatically source any file called `my.tcl` in the home directory. This file can contain `source` commands to define one's own procedures.

9.1.2 Command or Function?

Before writing a procedure to perform some Nap task, one needs to decide whether it is to be called as a command or as a function. The first question to ask is 'Is the sole purpose to define a single NAO?". If the answer is 'no' then it should be a command. If the answer is 'yes' then it should probably be a function, provided the arguments are not too complex. If there are many optional string arguments then a command would probably be better. Such a command can be called from within a Nap expression using the Tcl bracket ('`[ ]`') facility.

Note that multiple result NAOs can be combined into a single NAO using the *link* operator ('`,`') documented in section 5.3.3. The caller can separate the parts using function `open_box(`x`)`, as discussed in section 5.5.12.1.

9.1.3 Writing a Procedure to be called as a Function

The following examples comprise a variety of function definitions starting from the simplest imaginable and ending with some sophistication.

9.1.3.1 Function `sind`

Let's begin with a simple function defined by a simple expression with one argument. How about the sine of an angle in degrees? Let's call it '`sind`'. The procedure can be defined on one line as

follows:

```
% proc sind degrees {nap "sin(1r180p * degrees)"}
```

Note that '`1r180p`' is the constant $\pi/180$. Now let's test function '`sind`':

```
% nap "x = 0 .. 180 ... 30"
::NAP::76-76
% nap "y = sind x"
::NAP::83-83
% [nap "transpose(x /// y)"]
            0             0
           30           0.5
           60     0.8660254
           90             1
          120     0.8660254
          150           0.5
          180  1.224606e-16
```

9.1.3.2 Function `lam`

Now let's define a function (with two arguments x and y) defined by the above expression "`transpose(x /// y)`'.
This is the transpose of the *laminated* arguments, so let's call it '`lam`'.

```
% proc lam {
    x
    y
} {
    nap "z = x /// y"
    nap "transpose z"
}
```

There are two lines in the body of this procedure. The result of the final line defines the result of the function. Testing:

```
% [nap "lam(x,y)"]
            0             0
           30           0.5
           60     0.8660254
           90             1
          120     0.8660254
          150           0.5
          180  1.224606e-16
```

9.1.3.3 Function `get_bin`

Now let's define a function '`get_bin`' for binary input using the '`nap_get`' command:

```
% proc get_bin {
    filename
    {datatype {'f32'}}
    {swap 0}
} {
    # convert all arguments to strings
    set filename [[nap "filename"]]
    set datatype [[nap "datatype"]]
    set swap     [[nap "swap"]]
    set channel [open $filename]
```

```
    nap "in = [nap_get [lindex {binary swap} $swap] $channel $datatype]"
    close $channel
    nap "in"; # Define result
}
```

Note that the arguments '`datatype`' and '`swap`' have default values. Also note how all three arguments are converted from Nap expressions to Tcl strings.

Now let's test it. The following uses the OOC `binary` method to write six `f64` values to the file '`double.dat`'. Then this file is read using function '`get_bin`'.

```
% set file [open double.dat w]
filee1eb10
% [nap "{1.5 -3 0 2 4 5}"] binary $file
% close $file
% nap "x = get_bin('double.dat', 'f64')"
::NAP::27-27
% $x all
::NAP::27-27  f64  MissingValue: NaN  References: 1
Dimension 0   Size: 6      Name: (NULL)    Coordinate-variable: (NULL)
Value:
1.5 -3 0 2 4 5
```

9.1.3.4 Function `fact`

Now let's define a factorial function called '`fact`'. Of course we cannot resist the temptation to use recursion:

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

This works fine for scalar arguments:

```
% [nap "fact 4"]
24
% [nap "fact 1"]
1
% [nap "fact 0"]
1
```

But the following shows that it fails for a vector argument!

```
% [nap "fact {0 1 4 6}"]
1
```

9.1.3.5 Function `factorial`

One can define a proper elemental factorial function as follows:

```
% proc factorial n {
    if {[[nap "max(reshape(n)) > 1"]]} {
        nap "n > 1 ? n * factorial(n-1) : 1"
    } else {
        nap "1"
    }
```

```
}
% [nap "factorial {0 1 4 6}"]
1 1 24 720
```

Note the double brackets in the if command. The inner brackets produce an OOC-name. The outer brackets execute this OOC to produce the string '`0`' or '`1`'.

9.1.4 How Nap Functions Work

As an example, consider the expression '`a(b)`', which is of course equivalent to '`a b`'. Nap checks whether '`a`' is a Tcl variable. If not, it is assumed to be a function. In this case Nap first looks for a Tcl procedure called '`::NAP::a`." If this does not exist then Nap looks for a built-in Nap function called '`a`'. If this does not exist then Nap looks for a Tcl procedure called '`a`'.

The following example shows that a procedure with the global name '`sin`' does not override the built-in function with that name, whereas defining it within the Nap namespace '`::NAP::`' does override:

```
% proc sin x {nap "2*x"}
% [nap "sin 1"]
0.841471
% proc ::NAP::sin x {nap "2*x"}
% [nap "sin 1"]
2
```

It is possible to call some procedures as either functions or commands. The following example defines and uses the same function '`sind`' defined above:

```
% proc sind degrees {nap "sin(1r180p * degrees)"}
% [nap "sind 30"]; # call as function
0.5
% nap "s = [sind 30]"; # call as command within NAP
expression
::NAP::80-80
% $s
0.5
% [sind 30] all; # call as direct OOC
::NAP::86-86  f64  MissingValue: NaN  References: 0
Value:
0.5
```

But there is a problem calling procedures as commands if the result is referenced by a variable which is local to the procedure. At the end of the procedure Tcl deletes such local variables. This causes the referenced NAOs to be deleted. For example we could redefine function '`sind`' as follows:

```
% proc sind degrees {
    nap "result = sin(1r180p * degrees)"
    nap "result"
}
% [nap "sind 30"]
0.5
% [sind 30]
invalid command name "::NAP::32-32"
```

Note that the call as a function still worked but not the call as a command. Nap operates in a special mode while executing a procedure called as a function. The deletion of NAOs referenced by local variables is delayed until after the result has been saved. This is one advantage of calling procedures as functions rather than commands.

9.1.5 Writing a Procedure to be called as a Command

9.1.5.1 Command `write_expr`

First let's define a procedure whose Tcl result is empty and of no interest. It is obvious that such a procedure cannot be called as a function. The purpose of the procedure is to write to a text file the result of a Nap expression, which can of course contain variables and therefore must be executed in the caller's namespace. The following defines and tests the procedure:

```
% proc write_expr {
    expr
    filename
} {
    set channel [open $filename w]
    puts $channel [[uplevel nap \"$expr\"] value]
    close $channel
}
% nap "to = 5"
::NAP::52-52
% write_expr "1 .. to /// {0 7}" matrix.txt
% cat matrix.txt; # display contents of file 'matrix.txt'
1 2 3 4 5
0 7 0 7 0
```

9.1.5.2 Command `get_binary`

Next let's define a procedure called '`get_binary`' which is intended to be called as a command, but does essentially the same thing as the above function '`get_bin`'. This will help us to compare the two techniques in a situation where each has some advantages and some disadvantages. We assume the file '`double.dat`' still exists. The following example defines and tests procedure '`get_binary`':

```
% proc get_binary {
    filename
    {datatype f32}
    {swap 0}
} {
    set channel [open $filename]
    nap "in = [nap_get [lindex {binary swap} $swap] $channel $datatype]"
    close $channel
    nap "+in"; # Define result as copy of 'in' to prevent premature deletion
}
% nap "x = [get_binary double.dat f64]"
::NAP::63-63
% $x all
::NAP::63-63  f64  MissingValue: NaN  References: 1
Dimension 0   Size: 6      Name: (NULL)    Coordinate-variable: (NULL)
Value:
1.5 -3 0 2 4 5
```

Note that '`get_binary`' is simpler to define and simpler to use than '`get_bin`'. The main reason for this is the fact that all three arguments are used as strings rather than NAOs. One disadvantage of the command approach is the need to define the result as '`+in`' rather than simply '`in`'.

9.2 Interfacing Nap to a DLL based on C or Fortran Code

9.2.1 Introduction

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 be used either directly or via another interface (written in Tcl) defining a Nap function.

9.2.2 `make_dll`

The standard command used to create a DLL is:
`make_dll` *options newCommand argDec argDec argDec ...*
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` *string*: 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` *n.m*: Version number (default: `1.0`)

9.2.2.1 C Example

The following example (under SunOS 5.8) 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/sol/home/dav480/tcl/include -c pprod_i.c
ld -G -o libpprod.so pprod_i.o pprod.o
% load ./libpprod.so
% proc partialProd x {
    nap "result = reshape(f32(_), shape(x))"
    pprod  "nels(x)" x result
    nap "result"
}
% [nap "partialProd({2 1.5 3 0.5})"]
2 3 9 4.5
```

9.2.2.2 Fortran 90 Example

The following fortran 90 example does the same thing as the above C example. The f90 source code in the file `pprod.f90` is:

```
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 SunOS 5.8 f95 compiler. (Note that the entry point is '`pprod_`'.)

```
% exec f95 -c pprod.f90
% make_dll -entry pprod_ pprod {n i32 in} {x f32} {y f32 inout}
cc -I/sol/home/dav480/tcl/include -c pprod_i.c
ld -G -o libpprod.so pprod_i.o pprod.o
% load ./libpprod.so
% proc partialProd x {
    nap "result = reshape(f32(_), shape(x))"
    pprod  "nels(x)" x result
    nap "result"
}
% [nap "partialProd({2 1.5 3 0.5})"]
2 3 9 4.5
```

9.2.3 `make_dll_i`

This procedure is normally used via `make_dll`, but may be called directly if you prefer to do your own compiling and linking. It makes the Nap C interface to the user's C function or Fortran subroutine. It is called as follows:
`make_dll_i` *options newCommand argDec argDec argDec* ...
This command's result is the C code.

The arguments are similar to `make_dll`, except that the only options are:

`-entry` *string*: User-routine entry-point (default: *newCommand*). Note that fortran entry points often include suffix '_'.

`-header` *fileName*: header (`*.h`) filename (default: none)

`-version` *n.m*: Version number (default: `1.0`)

Chapter 10

Tcl Library Procedures called as Nap Functions

10.1 Introduction

The Nap installation process installs a library of Tcl procedures in directory *tcl_root*`/lib/nap*.*`. Many of these procedures define Nap functions, augmenting the built-in functions. This chapter describes these library functions defined with Tcl code.

10.2 Functions for Dates and Times

10.2.1 Introduction

The calendar commonly used in the modern world is the Gregorian Calendar introduced by Pope Gregory XIII in 1582. Years divisible by 400 (e.g. 1600, 2000) are leap years but other years divisible by 100 (e.g. 1800, 1900) are not. The Gregorian Calendar replaced the Julian Calendar introduced by Julius Caesar in 46 BC.

A *date* is defined by a *year* (AD for these functions), a *month-of-year* and a *day-of-month*. The following are examples of date problems:
How many days are there between two specified dates?
What is the date n days after a specified date?
What day of the week corresponds to a specified date?
Such problems can be solved by converting both ways between dates and the (integer) number of days since some fixed date.

Let us call a combination of date and time of day a *date/time*. The following are examples of date/time problems:
How many seconds are there between two specified date/times?
What is the date/time t seconds after a specified date/time?
Such problems can be solved by converting both ways between date/time and the time (as a real number of say days or seconds) since some fixed date/time.

The XXIIIrd International Astronomical Union General Assembly (1997) Resolution B1 recommends and defines (using different wording) the following terms:

Julian Day Number (JDN) integer assigned to a solar day, counting from 0 assigned to the day starting at noon UTC on 1st January 4713 BC (Julian Calendar).

Julian Date (JD) JDN plus the fraction of the day since the preceding noon UTC.

Modified Julian Date (MJD) time (in days) since 0000 hours UTC on 17th November 1858. MJD = JD − 2400000.5.

This astronomical definition of JDN can be relaxed for civil date calculations. Here one simply requires a way to associate a date with an integer. So the JDN can be used for civil dates commencing at local midnight. Alternatively, one can use the MJD corresponding to the preceding midnight. There is a difference of 2400001 between these integer JDN and MJD values used to identify entire days (see example below). Current dates correspond to a 7-digit JDN, as shown in the examples below.

The file `date.tcl` defines the four functions `date2jdn`, `jdn2date`, `dateTime2mjd` and `mjd2dateTime`. These are valid for all Gregorian dates from 1st January, 1 AD. The file `date.tcl` also defines the two formatting procedures `format_jdn` (see section 11.2.2) and `format_mjd` (see section 11.2.3).

Function `date2jdn` converts Gregorian dates to JDNs.
Function `jdn2date` converts JDNs to Gregorian dates.

Function `dateTime2mjd` converts Gregorian date/times to MJDs.
Function `mjd2dateTime` converts MJDs to Gregorian date/times.
Double-precision (64-bit) floating point gives accuracy of about a microsecond for years around 2000 AD.

These functions do not the handle the leap seconds that occur in UTC. These functions simply assume that each day contains exactly 86400 seconds (as in UT1). This can cause errors of several seconds in UTC time differences. This problem can be solved using a table of leap seconds such as that at the end of Resolution B1.

10.2.2 Dates and Julian Day Numbers (JDNs)

10.2.2.1 `date2jdn`(*ymd*)

This returns the JDNs of the specified Gregorian dates. The unit is set to `JDN`.

The argument *ymd* is an array whose final dimension has the size 3.
Column 0 contains the year AD (positive integer).
Column 1 contains the month of year (1 for Jan, 2 for Feb, ...).
Column 2 contains the day of month (0 to 31). Final example below illustrates 0.

Examples:

```
% [nap "date2jdn{1997 3 21}"] all; # 21st March, 1997. Note unit
::NAP::112-112  i32  MissingValue: -2147483648  References: 0  Unit: JDN
Value:
2450529
% [nap "date2jdn{1 1 1}"]; # 1st January, 1 AD
1721426
% [nap "date2jdn{1858 11 16}"]; # 16th November, 1858 AD
2400000
% [nap "date2jdn{{1997 2 28}{1997 3 1}}"]; # 28th February and 1st March, 1997
2450508 2450509
% [nap "date2jdn{{2004 8 31}{2004 9 0}}"]; # Both 31st August 2004
2453249 2453249
```

The following example calculates the day of the week for the first ten days of September 2004:

```
% [nap "ymd = transpose(2004 // (9 /// 1 .. 10))"]; # 1st to 10th Sept 2004
2004    9    1
2004    9    2
2004    9    3
2004    9    4
2004    9    5
2004    9    6
2004    9    7
2004    9    8
2004    9    9
2004    9   10
% [nap "date2jdn(ymd) % 7"] value; # 0 = Mon, 1 = Tue, 2 = Wed, ..., 6 = Sun
```

```
2 3 4 5 6 0 1 2 3 4
% [nap "1 + date2jdn(ymd) % 7"] value; # 1 = Mon, 2 = Tue, ..., 7 = Sun
3 4 5 6 7 1 2 3 4 5
% [nap "1 + (1 + date2jdn(ymd)) % 7"] value; # 1 = Sun, 2 = Mon, ..., 7 = Sat
4 5 6 7 1 2 3 4 5 6
```

10.2.2.2 jdn2date(*jdn*)

This is the inverse of function `date2jdn` above. It converts JDNs to Gregorian dates. The argument *jdn* is an array of JDNs. The shape of the result is
`shape(`*jdn*`) // 3`
where the final dimension (3) corresponds to year, month, day. The result is {_ _ _} for any date prior to 1st January, 1 AD (JDN = 1721426).

The following examples are the inverses of the first four examples above for `date2jdn`:

```
% [nap "jdn2date(1721426)"]
1 1 1
% [nap "jdn2date(2400000)"]
1858 11 16
% [nap "jdn2date(2450529)"]
1997 3 21
% [nap "jdn2date{2450508 2450509}"]
1997    2   28
1997    3    1
```

10.2.3 Date/Times and Modified Julian Dates (MJDs)

10.2.3.1 dateTime2mjd(*ymdhms*)

This returns the MJDs corresponding to the specified Gregorian date/times. The unit is set to `MJD`.

The argument *ymdhms* is an array whose final dimension has a size from 1 to 6.
Column 0 contains the year AD (positive integer)
Column 1 contains the month of year (1 for Jan, 2 for Feb, ...) (Default: 1)
Column 2 contains the day of month (0 to 31) (0: day before 1st) (Default: 1)
Column 3 contains the hour of day (0 to 23) (default: 0)
Column 4 contains the minute of hour (0 to 59) (default: 0)
Column 5 contains the (possibly fractional) second of minute (0 to 60) (default: 0)

Examples:

```
% [nap "dateTime2mjd{2004 8 16 14 39 59.5}"] all -f %.7f
::NAP::525-525  f64  MissingValue: NaN  References: 0  Unit: MJD
Value:
53233.6111053
% [nap "dateTime2mjd{
{1858 11 16}
{1858 11 17}
{1858 11 18}
}"]
-1 0 1
% [nap "dateTime2mjd{1 1 1}"];# 1st Jan 0001 AD
-678575
% [nap "dateTime2mjd{2000 1 0 12}"]; # 1200 hours, 31st December 1999
51543.5
# The following gives the difference between the JDN and MJD for a date
% [nap "date2jdn{2004 8 16} - dateTime2mjd{2004 8 16}"] -f %d
2400001
```

10.2.3.2 `mjd2dateTime`(*mjd*[, *delta*])

This is the inverse of function `dateTime2mjd` above. It converts MJDs to Gregorian date/times.

The argument *mjd* is an array of MJDs.

The optional argument *delta* controls rounding. The result is rounded to the nearest multiple of *delta* seconds. The default value is 1, which rounds to the nearest second. A value of 60 would round to the nearest minute. A value of 1e-3 would round to the nearest millisecond.

The shape of the result is
`shape`(*mjd*) `// 6`
where the final dimension (6) corresponds to year, month, day, hour, minute, second.

The following examples are the inverses of the first three examples for `dateTime2mjd`:

```
% [nap "mjd2dateTime(53233.6111053, 0.1)"]; # round to multiple of 0.1 seconds
2004 8 16 14 39 59.5
% [nap "mjd2dateTime(-1 .. 1)"]
1858   11   16    0    0    0
1858   11   17    0    0    0
1858   11   18    0    0    0
% [nap "mjd2dateTime(-678575)"];# 1st Jan 0001 AD
1 1 1 0 0 0
```

10.2.4 Calendars with Fixed-length Years

Computer models (of climate etc.) sometimes use simplified calendars in which every year has the same number of days (e.g. 360, 365 or 366). The following two functions `dateTime2days` and `days2dateTime`, handle such calendars. In each case there is a 12-element vector argument specifying the number of days in each month of the year. Time is measured relative to the start of the base year (year 0) and can be negative.

10.2.4.1 `dateTime2days`(*ymdhms*, *ndim*)

This returns the time in days since the start of year 0. This function is similar to the above `dateTime2mjd` (see section 10.2.3.1) except that it uses the unconventional calendar defined by *ndim*.

The argument *ymdhms* is an array whose final dimension has a size from 1 to 6.
Column 0 contains the (possibly negative) integer year relative to year 0
Column 1 contains the month of year (1 for Jan, 2 for Feb, ...) (Default: 1)
Column 2 contains the day of month (0 to 31) (0: day before 1st) (Default: 1)
Column 3 contains the hour of day (0 to 23) (default: 0)
Column 4 contains the minute of hour (0 to 59) (default: 0)
Column 5 contains the (possibly fractional) second of minute (0 to 60) (default: 0)

The argument *ndim* is a 12-element vector giving *number of days in month*.

Examples:

```
% nap "ndim = {31 28 31 30 31 30 31 31 30 31 30 31}"; # number of days in month
% [nap "dateTime2days({1 1 1 18}, ndim)"]
365.75
% [nap "dateTime2days({{1 1 1}{1 4 10}{0 1 1}{-1 1 1}}, ndim)"]
365 464 0 -365
```

10.2.4.2 `days2dateTime`(*days*, *ndim*[, *delta*])

This converts a time (in days since the start of year 0) to a calendar date and time-of-day. This function is similar to the above `mjd2dateTime` (see section 10.2.3.2) except that it uses the unconventional calendar defined by *ndim*.

The argument *days* is the (possibly negative) time in days since the start of the year 0.

The argument *ndim* is a 12-element vector giving *number of days in month*.

The optional argument *delta* controls rounding. The result is rounded to the nearest multiple of *delta* seconds. The default value is 1, which rounds to the nearest second. A value of 60 would round to the nearest minute. A value of 1e-3 would round to the nearest millisecond.

Examples:

```
% nap "ndim = {31 28 31 30 31 30 31 31 30 31 30 31}"; # number of days in month
% [nap "days2dateTime(365.75, ndim)"]
1 1 1 18 0 0
% [nap "days2dateTime({365 464 0 -365}, ndim)"]
 1  1  1  0  0  0
 1  4 10  0  0  0
 0  1  1  0  0  0
-1  1  1  0  0  0
```

10.3 Statistical Functions

10.3.1 Introduction

These are defined in `stat.tcl`. Note that there are also built-in statistical functions called `correlation` and `moving_correlation`, which are described in section 5.5.8. The *tally* unary operator '`#`' (see section 5.3.7) also has statistical applications.

10.3.2 Simple Statistics

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 6}"
```

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

```
% nap "m = {
    {1.5 2.1 1.5 0.2}
    {6.2 4.9   _ 0.2}
}"
```

10.3.2.1 Arithmetic mean: `am`(x[, *verb_rank*])

```
% [nap "am(v)"]
9.625
% [nap "am(m)"]
3.85 3.5 1.5 0.2
% [nap "am(m,1)"]
1.325 3.76667
```

10.3.2.2 Coefficient of variation (with division by n): `CV`(x[, *verb_rank*])

This is the uncorrected standard deviation (`sd`(x)) divided by the arithmetic mean.

```
% [nap "CV(v)"]
0.495383
% [nap "CV(m)"]
0.61039 0.4 0 0
% [nap "CV(m,1)"]
0.523904 0.684226
```

10.3.2.3 Coefficient of variation (with division by $n-1$): `CV1(`x`[, `*verb_rank*`])`

This is the corrected standard deviation (`sd1(`x`)`) divided by the arithmetic mean.

```
% [nap "CV1(v)"]
0.529586
% [nap "CV1(m)"]
0.863221 0.565685 _ 0
% [nap "CV1(m,1)"]
0.604952 0.838002
```

10.3.2.4 Geometric mean: `gm(`x`[, `*verb_rank*`])`

```
% [nap "gm(v)"]
8.38752
% [nap "gm(m)"]
3.04959 3.2078 1.5 0.2
% [nap "gm(m,1)"]
0.985957 1.82476
```

10.3.2.5 Median: `median(`x`[, `*verb_rank*`])`

```
% [nap "median(v)"]
8.5
% [nap "median(m)"]
3.85 3.5 1.5 0.2
% [nap "median(m,1)"]
1.5 4.9
```

10.3.2.6 Mode: `mode(`x`[, `*verb_rank*`])`

```
% [nap "mode(v)"]
6
% [nap "mode(m)"]
3.85 3.5 1.5 0.2
% [nap "mode(m,1)"]
1.50625 3.7625
```

10.3.2.7 Percentiles: `percentile(`x`, `*pc*`[, `*verb_rank*`[, `*nc*`]])`

pc = vector of required percentiles
nc = number of class intervals (default: 256)

The following examples calculate the following percentiles:

- 0 (minimum)
- 50 (median)
- 100 (maximum)

```
% [nap "percentile(v, {0 50 100})"]
3 8.50781 18
% [nap "percentile(m, {0 50 100})"]
1.50000 2.10000 1.50000 0.20000
3.85918 3.50547 1.50000 0.20000
6.20000 4.90000 1.50000 0.20000
% [nap "percentile(m, {0 50 100}, 1)"]
0.20000 0.20000
1.50254 4.92266
2.10000 6.20000
```

10.3.2.8 Root mean square: `rms`(*x*[, *verb_rank*])

```
% [nap "rms(v)"]
10.7413
% [nap "rms(m)"]
4.51054 3.76962 1.5 0.2
% [nap "rms(m,1)"]
1.49583 4.56399
```

10.3.2.9 Standard-deviation (with division by n): `sd`(*x*[, *verb_rank*])

```
% [nap "sd(v)"]
4.76806
% [nap "sd(m)"]
2.35 1.4 0 0
% [nap "sd(m,1)"]
0.694172 2.57725
```

10.3.2.10 Standard-deviation (with division by $n-1$): `sd1`(*x*[, *verb_rank*])

```
% [nap "sd1(v)"]
5.09727
% [nap "sd1(m)"]
3.3234 1.9799 _ 0
% [nap "sd1(m,1)"]
0.801561 3.15647
```

10.3.2.11 Variance (with division by n): `var`(*x*[, *verb_rank*])

```
% [nap "var(v)"]
22.7344
% [nap "var(m)"]
5.5225 1.96 0 0
% [nap "var(m,1)"]
0.481875 6.64222
```

10.3.2.12 Variance (with division by $n-1$): `var1`(*x*[, *verb_rank*])

```
% [nap "var1(v)"]
25.9821
% [nap "var1(m)"]
11.045 3.92 _ 0
% [nap "var1(m,1)"]
0.6425 9.96333
```

10.3.3 Moving Average: `moving_average`(*x*, *shape_window*[, *step*])

Move window of specified shape by specified step (can vary for each dimension). Result is arithmetic mean of x values in each window. x can have any rank > 0.

shape_window is either a scalar or a vector with an element for each dimension. If it is a scalar then it is treated as a vector with `rank`(x) identical elements.

Similarly, *step* is either a scalar or a vector with an element for each dimension. If it is a scalar then it is treated as a vector with `rank`(x) identical elements. The default value for *step* is 1. If *step* has the same value as *shape_window* then the windows do not overlap. The value -1 is treated like 1, except that missing values are prepended and appended (along this dimension of x) to produce a result with the same dimension size as x.

The following examples illustrate the application of `moving_average` to a vector.

```
% nap "v = {12 6 _ 7 3 15 _ 10 18 5}"
% [nap "moving_average(v, 3)"] value
9 6.5 5 8.33333 9 12.5 14 11
% [nap "moving_average(v, 3, 3)"] value
9 8.33333 14
% [nap "moving_average(v, 3, -1)"] value
9 9 6.5 5 8.33333 9 12.5 14 11 11.5
```

The following examples illustrate the application of `moving_average` to a matrix.

```
% nap "m = {
    {1 2 1 4 0 0}
    {5 4 2 9 2 7}
    {0 1 1 3 1 4}
}"
% [nap "moving_average(m, {3 3})"] value
1.88889 3.00000 2.55556 3.33333
% [nap "moving_average(m, {3 3}, {3 3})"] value
1.88889 3.33333
% [nap "moving_average(m, 3, 3)"] value
1.88889 3.33333
% [nap "moving_average(m, 3, -1)"] value
3.00000 2.50000 3.66667 3.00000 3.66667 2.25000
2.16667 1.88889 3.00000 2.55556 3.33333 2.33333
2.50000 2.16667 3.33333 3.00000 4.33333 3.50000
% [nap "moving_average(m, {1 3})"] value
1.333333 2.333333 1.666667 1.333333
3.666667 5.000000 4.333333 6.000000
0.666667 1.666667 1.666667 2.666667
```

10.3.4 Least Squares Regresssion and Curve Fitting

Function `regression` provides the coefficients of a least squares linear multiple regression equation. Function `fit_polynomial` provides the coefficients of a least squares polynomial of specified order. Function `polynomial` evaluates a polynomial.

10.3.4.1 `regression(`x`, `y`)`

This function produces an array containing the coefficients for the regression of y on x. In other words it defines an equation for predicting y from x.

The arrays x and y can be vectors or a matrices. Vectors are treated as matrices with a single column. The columns of x correspond to the known variables. The columns of y (and the result) correspond to the predicted variables. x and y must have the same number of rows. Each row corresponds to a case (e.g. person).

If x and y are both vectors (of the same length) then the result is the 2-element vector a which defines the simple linear regression equation of Y on X given by

$$Y = a_0 + a_1 X$$

The following example of such simple linear regression is from page 248 of *Schaum's Outline of Theory and Problems of Statistics*, M.R. Spiegel, 1961. The heights of fathers are used to predict the heights of their eldest sons.

```
% nap "x = {65 63 67 64 68 62 70 66 68 67 69 71}"; # height (inches) of 12 fathers
::NAP::14-14
% nap "y = {68 66 68 65 69 66 68 65 71 67 68 70}"; # height (inches) of their sons
::NAP::16-16
% [nap "regression(x, y)"]
35.8248 0.476378
```

Thus the regression line of Y on X is

$$Y = 35.8248 + 0.476378X$$

Consider the case where x is an $m \times n$ matrix and y is an m-element vector. The sample size (number of cases) is m. The result is the $(n+1)$-element vector a which defines the linear multiple regression equation

$$Y = a_0 + a_1X_0 + a_2X_1 + a_3X_2 + \cdots + a_nX_{n-1}$$

which predicts a single variable Y from the n variables $X_0, X_1, X_2, \ldots, X_{n-1}$. If we are predicting a single case then these variables can be combined into an n-element vector X. If we are predicting M cases then X is an $M \times n$ matrix. Thus the above equation can be expressed in matrix form as

$$Y = (1 \| X) \cdot a$$

where '$\|$' is the *concatenate* operator (prepending to the matrix X a column of ones which essentially defines a pseudo-variable X_{-1} which is multiplied by a_0) and '$\cdot$' is the *matrix-multiplication* operator. Note that the above equation is valid for vector X provided we then (as in *nap*) treat '$\cdot$' as the *vector dot (scalar) product* operator.

The following example of multiple linear regression is from page 273 of *Schaum's Outline of Theory and Problems of Statistics*, M.R. Spiegel, 1961. The ages and heights of boys are used to predict their weights.

```
% nap "age    = { 8 10  6 11  8  7 10  9 10  6 12  9}"; # years
::NAP::37-37
% nap "height = {57 59 49 62 51 50 55 48 52 42 61 57}"; # inches
::NAP::39-39
% nap "weight = {64 71 53 67 55 58 77 57 56 51 76 68}"; # pounds
::NAP::41-41
% nap "x = transpose(age /// height)"
::NAP::44-44
% [nap "b = regression(x, weight)"]
3.65122 1.50633 0.85461
```

Thus the least squares regression equation of weight w on age a and height h is

$$w = 3.65122 + 1.50633a + 0.85461h$$

Let us use this equation to predict the weight of a boy who is 9 years old and 54 inches tall.

```
% [nap "{1 9 54} . b"]
63.3571
```

The predicted weight is 63.3571 pounds.

Next let us use this equation to predict the weights of three boys. The first is again 9 years old and 54 inches tall. The second is 6 years old and 40 inches tall. The third is 12 years old and 60 inches tall.

```
% [nap "{{1 9 54}{1 6 40}{1 12 60}} . b"]
63.3571 46.8736 73.0038
```

Thus the predicted weights are 63.3571, 46.8736 and 73.0038 respectively.

Now consider the case where we are predicting multiple variables from a single variable. Using the above example, let us predict both height and weight from age. We could use two separate commands as in the following:

```
% [nap "regression(age, height)"]
31.5 2.5
% [nap "regression(age, weight)"]
30.5714 3.64286
```

Thus the desired regression equations (predicting height h and weight w from age a) are

$$\begin{aligned} h &= 31.5 + 2.5a \\ w &= 30.5714 + 3.64286a \end{aligned}$$

Now we can use these equations to predict the heights and weights of boys aged 6 and 10:

```
% [nap "31.5 + 2.5 * {6 10}"]
46.5 56.5
% [nap "30.5714 + 3.64286 * {6 10}"]
52.4286 67
```

Thus a boy aged 6 is expected to have a height of 46.5 inches and a weight of 52.4 pounds, while a boy aged 10 is expected to have a height of 56.5 inches and a weight of 67 pounds.

The following example shows how we can define and use both equations in a single expression:

```
% [nap "{{1 6}{1 10}} . regression(age, transpose(height /// weight))"]
46.5000 52.4286
56.5000 67.0000
```

Here x (the first argument of function *regression*) is the vector *age* and y (the second argument of function *regression*) is the matrix consisting of the column vectors *height* and *weight*. Let X be the vector containing the values of x for which predictions are required. Now the desired predictions are again given by the matrix equation

$$Y = (1 \| X) \cdot a$$

In fact this equation is always valid, including the case where both x and y are matrices.

10.3.4.2 `fit_polynomial`(x, y, n) and `polynomial`(c, x)

Function `fit_polynomial`(x, y, n) provides the coefficients of a least squares polynomial of order n (default 1, giving straight line). x and y are vectors of the same length.

The following example uses the *age* and *height* vectors defined above to define the least-squares parabola that predicts *height* from *age*.

```
% [nap "c = fit_polynomial(age, height, 2)"]; # Define coefficients of parabola
28.8273 3.13801 -0.0364315
```

Thus the parabola is $28.8273 + 3.13801a - 0.0364315a^2$

The following uses this parabola to again estimate height at age 6 and 10.

```
% nap "a = {6 10}"; # ages in years
::NAP::56-56
% [nap "c(0) + c(1) * a + c(2) * a ** 2"]; # predicted height at ages 6 & 10
46.3439 56.5643
```

Thus the height at age 6 is estimated to be 46.3439 inches rather than the previous linear estimate of 46.5 inches. And the height at age 10 is estimated to be 56.5643 inches rather than the previous linear estimate of 56.5 inches.

Function `polynomial`(c, x) evaluates the polynomial defined by the vector of coefficients c (e.g. as given by function `fit_polynomial`). x can have any shape. The result has the same shape as that of x.

The following example uses function `polynomial` to evaluate this parabola again.

```
% [nap "polynomial(c, a)"]; # predicted height at ages 6 & 10
46.3439 56.5643
```

10.4 Geographic Functions

10.4.1 Introduction

Except where otherwise stated, the following functions are defined in the file `geog.tcl`. Note that this file also defines geographic *procedures*, as described in section 11.4.

10.4.2 Divergence and Vorticity of Wind

Functions `div_wind(`u`, `v`)` and `vorticity_wind(`u`, `v`)` give the divergence (second^{-1}) and vorticity (second^{-1}) of a 2D wind, where
u is matrix containing zonal (x-component i.e. from west to east) wind in metres/sec.
v is matrix containing meridional (y-component i.e. from south to north) wind in metres/sec.
Coordinate variables of u and v are latitude (°N) and longitude (°E).

Let θ be latitude in radians, ϕ be longitude in radians, r be radius of earth in metres.

The divergence of a 2D wind is defined as

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}$$

Function `div_wind` uses the equivalent formula

$$\frac{\frac{\partial u}{\partial \phi} + \frac{\partial (v \cos\theta)}{\partial \theta})}{r cos\theta}$$

The vorticity of a 2D wind is defined as

$$\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}$$

Function `vorticity_wind` uses the equivalent formula

$$\frac{\frac{\partial v}{\partial \phi} - \frac{\partial (u \cos\theta)}{\partial \theta})}{r cos\theta}$$

10.4.3 Things related to Latitude and Longitude

10.4.3.1 `area_on_globe(`*latitude*`, `*longitude*`)`

Given latitude and longitude vectors, this function calculates a matrix whose values are fractions of the Earth's surface area corresponding to cells defined by the functions `merid_bounds(`*longitude*`)` (see section 10.4.3.3) and `zone_bounds(`*latitude*`)` (see section 10.4.3.5).

Example:

```
% [nap "lats = 90 .. -90 ... -45"]
90 45 0 -45 -90
% [nap "lons = -180 .. 180 ... 45"] value
-180 -135 -90 -45 0 45 90 135 180
% [nap "area = area_on_globe(lats, lons)"] value -format %.4f
0.0046 0.0092 0.0092 0.0092 0.0092 0.0092 0.0092 0.0092 0.0046
0.0156 0.0313 0.0313 0.0313 0.0313 0.0313 0.0313 0.0313 0.0156
0.0221 0.0442 0.0442 0.0442 0.0442 0.0442 0.0442 0.0442 0.0221
0.0156 0.0313 0.0313 0.0313 0.0313 0.0313 0.0313 0.0313 0.0156
0.0046 0.0092 0.0092 0.0092 0.0092 0.0092 0.0092 0.0092 0.0046
```

10.4.3.2 `fix_longitude(`*longitude*`)`

Adjust elements of longitude vector by adding or subtracting multiple of 360 to ensure:

$$-180 \le x_0 < 180$$

$$0 \le x_{i+1} - x_i < 360$$

Also ensure unit is `degrees_east`.

10.4.3.3 `merid_bounds`(*longitude*)

This function defines reasonable boundary longitudes from a vector giving the central longitudes of each cell. The range of longitudes should not exceed 360° but may be much smaller.

The result is a vector with one more element than the argument. The midpoints between adjacent argument values define all the elements of the result except the first and final ones.

The first and final values are defined as follows. Trial outer cells are added with the same width as their adjacent cell. If these outer cells overlap (due to global wrap-around) then both outer boundaries are set to the value (plus an integer multiple of 360°) midway between the first and final values of the argument.

Examples:

```
% [nap "merid_bounds {-180 -90 0 30 90 180}"] value; # global-range
-180 -135 -45 15 60 135 180
% [nap "merid_bounds {10 30 40 50 60}"] value; # local-range
0 20 35 45 55 65
```

10.4.3.4 `merid_wt`(*longitude*)

Calculate normalised (so sum of weights is 1) meridional weights from longitudes. These weights are proportional to the widths of the cells defined by the above (section 10.4.3.3) function `merid_bounds`(*longitude*).

Examples:

```
% [nap "merid_wt {-180 -90 0 30 90 180}"] value; # global-range
0.125 0.25 0.166667 0.125 0.208333 0.125
% [nap "merid_wt {10 30 40 50 60}"] value; # local-range
0.307692 0.230769 0.153846 0.153846 0.153846
```

10.4.3.5 `zone_bounds`(*latitude*)

This function defines reasonable boundary latitudes from a vector giving the central latitudes of each cell.

The result is a vector with one more element than the argument. If the argument is an arithmetic progression (AP) then the midpoints between adjacent argument values define all the elements of the result except the first and final ones, otherwise each of these boundaries separates two equal *surface areas* between adjacent argument values. The two outer boundaries are defined using steps (in latititude or area) equal to their adjacent step, with the obvious limit in magnitude of 90°.

Examples:

```
% [nap "zone_bounds(-80 .. 80 ... 40)"] value; # global-range AP
-90 -60 -20 20 60 90
% [nap "zone_bounds(90 .. -90 ... -30)"] value; # global-range AP
90 75 45 15 -15 -45 -75 -90
% [nap "zone_bounds{-80 -40 -10 10 40 80}"] value; # global-range non-AP
-90 -54.4687 -24.0929 0 24.0929 54.4687 90
% [nap "zone_bounds(10 .. 60 ... 10)"] value; # local-range AP
5 15 25 35 45 55 65
```

10.4.3.6 `zone_wt`(*latitude*)

Calculate normalised (so sum of weights is 1) zonal weights from latitudes. These weights are proportional to the surface areas of the cells (of longitude width 360°) defined by the above (section 10.4.3.5) function `zone_bounds`(*latitude*). If the Earth's radius is R then the area between two latitudes θ_1 and θ_2 is $2\pi R^2(\sin\theta_1 - \sin\theta_2)$.

Examples:

```
% [nap "zone_wt(-80 .. 80 ... 40)"] value; # global-range AP
0.0669873 0.262003 0.34202 0.262003 0.0669873
% [nap "zone_wt(90 .. -90 ... -30)"] value; # global-range AP
0.0170371 0.12941 0.224144 0.258819 0.224144 0.12941 0.0170371
% [nap "zone_wt{-80 -40 -10 10 40 80}"] value; # global-range non-AP
0.0931011 0.20279 0.204109 0.204109 0.20279 0.0931011
% [nap "zone_wt(10 .. 60 ... 10)"] value; # local-range AP
0.209562 0.199962 0.184286 0.16301 0.136782 0.106398
```

10.4.4 Land, Water and Shoreline

Functions `is_land`, `is_coast` and `fraction_land` produce grid results. Functions `acof2boxed` and `get_gshhs` produce boxed arrays of polyline shorelines.

10.4.4.1 Functions based on `land_flag`

The functions `is_land`, `is_coast` and `fraction_land` are based on the `nap_land_flag` command described in section 8.2. This uses data which has an accuracy of 0.01°.

These three functions are defined in the file `land.tcl`.

These functions take an optional argument *data_dir*, which specifies the directory containing the `nap_land_flag` data files. This argument is not needed when the standard data directory is used.

10.4.4.1.1 `is_land`(*latitude*, *longitude*[, *data_dir*])

Produce a land/sea mask in the form of an `i8` (8-bit signed integer) matrix with 1 for land and 0 for sea.

The arguments *latitude* and *longitude* can be scalars, vectors or matrices. If they are either both vectors, or one is a vector and the other is a scalar, then they are used as the coordinate variables of the matrix result. Otherwise the result has the same shape and coordinate variables as the one of higher rank.

Note that each element of the result corresponds to just the single point defined by its latitude and longitude. It is often better to test multiple points, which can be done using function `fraction_land`(*latitude*, *longitude*[, *nlat*, *nlon*, *data_dir*]) described in section 10.4.4.1.3.

The following example produces a matrix with `0` for sea and `1` for land:

```
% [nap "lats = 90 .. -90 ... -45"]
90 45 0 -45 -90
% [nap "lons = -180 .. 180 ... 45"] value
-180 -135 -90 -45 0 45 90 135 180
% [nap "isLand = is_land(lats, lons)"] value
0 0 0 0 0 0 0 0 0
0 0 1 0 1 1 1 1 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1
```

The following shows how this can be used to mask out (make missing) the sea points in the matrix named `area` defined in the the above example in section 10.4.3.1:

```
% [nap "isLand ? area : _"] value -format %.4f
     _      _      _      _      _      _      _      _      _
     _      _ 0.0313      _ 0.0313 0.0313 0.0313 0.0313      _
     _      _      _      _      _      _      _      _      _
     _      _      _      _      _      _      _      _      _
0.0046 0.0092 0.0092 0.0092 0.0092 0.0092 0.0092 0.0092 0.0046
```

The following masks out the land points:

```
% [nap "isLand ? _ : area"] value -format %.4f
0.0046 0.0092 0.0092 0.0092 0.0092 0.0092 0.0092 0.0092 0.0046
0.0156 0.0313      _ 0.0313      _      _      _      _ 0.0156
0.0221 0.0442 0.0442 0.0442 0.0442 0.0442 0.0442 0.0442 0.0221
0.0156 0.0313 0.0313 0.0313 0.0313 0.0313 0.0313 0.0313 0.0156
     _      _      _      _      _      _      _      _      _
```

The following alternative method defines a mask consisting of zeros and missing values. Masking is done by adding this mask.

```
% [nap "mask = {_ 0}(isLand)"] value
_ _ _ _ _ _ _ _ _
_ _ 0 _ 0 0 0 0 _
_ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _
0 0 0 0 0 0 0 0 0
% [nap "area + mask"] value -format %.4f
     _      _      _      _      _      _      _      _      _
     _      _ 0.0313      _ 0.0313 0.0313 0.0313 0.0313      _
     _      _      _      _      _      _      _      _      _
     _      _      _      _      _      _      _      _      _
0.0046 0.0092 0.0092 0.0092 0.0092 0.0092 0.0092 0.0092 0.0046
```

10.4.4.1.2 `is_coast`(*latitude*, *longitude*[, *nlat*, *nlon*, *data_dir*])

This function produces an `i8` (8-bit signed integer) matrix with 1 for coast and 0 otherwise.

Cells are defined around each latitude and longitude using the functions `zone_bounds`(*latitude*) (see section 10.4.3.5) and `merid_bounds`(*longitude*) (see section 10.4.3.3). Each cell is divided into *nlat* rows and *nlon* columns. A cell is considered land if more than half its *nlat* × *nlon* points are classified as land by function `is_land`.

The result is 1 if a cell is land and has at least one adjacent sea cell to the north, south, east or west. The coordinate variables of the result are copies of the vector arguments *latitude* and *longitude*.

The optional argument *nlat* defaults to 8.
The optional argument *nlon* defaults to *nlat*.
If both these arguments are omitted then each cell contains 64 points, which is enough for most purposes. (A single point is often adequate for shoreline overlays on maps, etc.)

10.4.4.1.3 `fraction_land`(*latitude*, *longitude*[, *nlat*, *nlon*, *data_dir*])

This function produces an `f32` (32-bit floating-point) matrix containing the estimated proportion of land in each cell.

Cells are defined around each latitude and longitude using the functions `zone_bounds`(*latitude*) (see section 10.4.3.5) and `merid_bounds`(*longitude*) (see section 10.4.3.3). Each cell is divided into *nlat* rows and *nlon* columns. The result is defined by the proportion of the *nlat* × *nlon* points which are classified as land by function `is_land`.

The coordinate variables of the result are copies of the vector arguments *latitude* and *longitude*.

The optional argument *nlat* defaults to 8. The optional argument *nlon* defaults to *nlat*. If both these arguments are omitted then each cell contains 64 points, which is enough for most purposes.

The minimum resolution is 0.01° even if the specified *nlat* or *nlon* would give less. The value of *nlat* or *nlon* is automatically reduced if its specified value would give a resolution less than 0.01°.

The following example produces a similar (but much more accurate) land/sea mask to that produced in the above example in section 10.4.4.1.1 for function `is_land`:

```
% [nap "lats = 90 .. -90 ... -45"]
90 45 0 -45 -90
% [nap "lons = -180 .. 180 ... 45"] value
-180 -135 -90 -45 0 45 90 135 180
% [nap "fraction_land(lats, lons)"] value -format %0.2f
```

```
0.05 0.14 0.28 0.55 0.08 0.17 0.34 0.19 0.11
0.06 0.39 0.67 0.13 0.52 0.88 1.00 0.50 0.17
0.00 0.00 0.22 0.38 0.47 0.45 0.17 0.23 0.00
0.00 0.00 0.09 0.13 0.03 0.05 0.00 0.25 0.02
0.25 0.59 0.69 0.47 0.81 0.95 0.95 0.98 0.47
% [nap "fraction_land(lats, lons) > 0.5f32"] value
0 0 0 1 0 0 0 0 0
0 0 1 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 1 1 0 1 1 1 1 0
```

10.4.4.2 `acof2boxed`(*filename*[, *min_longitude*])

Read *ACOF (ascii coastal outline file)* coastline file and return a boxed nao.

Each element of this boxed nao is a 2-row matrix. Row 0 contains longitudes. Row 1 contains latitudes. These coordinates define a polyline shoreline of an island, lake, etc.

The argument *min_longitude* (default: −180) specifies the desired minimum longitude. This is normally −180 (giving longitudes in the range −180 to 180) or 0 (giving longitudes in the range 0 to 360).

An ACOF file is a text file defining polyline shorelines. Each polyline commences with two header lines. The first of these contains arbitrary text (for name of Island, etc.) which function `acof2boxed` ignores. The second line contains the number of points in the polyline. Each point corresponds to a line containing a longitude and latitude.

The following is a listing of an ACOF file `nz_tas.acof` defining very simple shorelines for Tasmania and New Zealand:

```
Tasmania
       5
    143.44    -44.60
    143.44    -41.41
    149.06    -41.41
    149.06    -44.60
    143.44    -44.60
New Zealand
       7
    165.94    -44.60
    165.94    -41.41
    177.19    -41.41
    177.19    -38.23
    171.56    -38.23
    171.56    -44.60
    165.94    -44.60
```

The following example reads this file:

```
% nap "nz_tas = acof2boxed('nz_tas.acof', 0)"; # longitudes from 0 to 360
::NAP::425-425
% [nap "open_box(nz_tas(0))"] value
143.44 143.44 149.06 149.06 143.44
-44.60 -41.41 -41.41 -44.60 -44.60
% [nap "open_box(nz_tas(1))"] value
165.94 165.94 177.19 177.19 171.56 171.56 165.94
-44.60 -41.41 -41.41 -38.23 -38.23 -44.60 -44.60
```

The following example shows how `acof2boxed` can be used in conjunction with `plot_nao` to draw shorelines:

```
plot_nao u -overlay "E acof2boxed('nz_tas.acof')"
```

10.4.4.3 get_gshhs(*resolution*, *min_area*, *max_level*, *min_longitude*, *max_longitude*, *min_latitude*, *max_latitude*, *data_dir*)

Read *GSHHS* (Global Self-consistent Hierarchical High-resolution Shorelines) shoreline data and return a boxed nao.

Each element of this boxed nao result is a 2-row matrix. Row 0 contains longitudes. Row 1 contains latitudes. These coordinates define a polyline shoreline of an island, lake, etc.

Function `get_gshhs` is defined in the file `gshhs.tcl`.

GSHHS is distributed with the following data files:

Filename	Description	Resolution	File Size
`gshhs_c.b`	crude resolution	25 km	0.2 Mb
`gshhs_l.b`	low resolution	5 km	1.2 Mb
`gshhs_i.b`	intermediate resolution	1 km	5.3 Mb
`gshhs_h.b`	high resolution	0.2 km	21.4 Mb
`gshhs_f.b`	full resolution		90.3 Mb

Only the crude, low and intermediate resolution files are distributed with Nap. If higher resolution is required then copy the required files to the data directory before calling `get_gshhs`.

The arguments are:

resolution (default: 1): Required accuracy (km). This determines which data file is used.

min_area (default: 0): Exclude polygons whose area (square km) is less than *min_area*.

max_level (default: 4): Exclude polygons whose *level* > *max_level*. The value of *level* is 1 for *land*, 2 for *lake*, 3 for *island in lake* and 4 for *pond in island in lake*.

min_longitude (default: −180): Left boundary of region. Resultant longitudes range from *min_longitude* to *min_longitude* + *360*.

max_longitude (default: *min_longitude* + *360*): Right boundary of region.

min_latitude (default: −90): Bottom boundary of region.

max_latitude (default: 90): Top boundary of region.

data_dir: Directory containing GSHHS files with various resolutions. The default is *tcl_root*`/lib/nap*.*/data/gshhs` unless environment variable `GSHHS_DATA_DIR` is defined to overide this.

The following example shows how `get_gshhs` can be used in conjunction with `plot_nao` to draw shorelines:

```
plot_nao u -overlay "E get_gshhs(25)"
```

10.4.5 Reading Miscellaneous Geographic Files

10.4.5.1 get_gridascii(*filename*[, *unit*])

Read a file in ARC/INFO GRIDASCII format.

The optional argument *unit* specifies the unit of x and y. If this argument is omitted then:
Dimension x is longitude and has unit `"degrees_east"`.
Dimension y is latitude and has unit `"degrees_north"`.

Examples:

```
% nap "in = get_gridascii('abc.gridascii')"; # x is longitude, y is latitude
% nap "in = get_gridascii('abc.gridascii', 'metres')"
```

10.5 Miscellaneous Functions

10.5.1 Introduction

The following functions are defined in the file `nap_function_lib.tcl`.

10.5.2 color_wheel(n, v, b)

Square containing color wheel.
n is number rows and 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`} and values from 0 to 255.

10.5.3 `cpi(`*array*`[,`*i*`[,`*j*`[,`*k* `...]]])`

Cross-product indexing without the need to specify commas for trailing dimensions that are to be fully selected. This is useful when the rank of the array can vary.

Examples:

```
% [nap "m = {{1 3 5}{6 8 9}}"]
1 3 5
6 8 9
% [nap "cpi(m, {1 0})"]
6 8 9
1 3 5
% [nap "cpi(m, 1, {2 0})"]
9 6
% [nap "cpi(m, , 1)"]
3 8
% [nap "cpi(m,0,)"]
1 3 5
% [nap "cpi(m,0)"]
1 3 5
% [nap "cpi(m)"]
1 3 5
6 8 9
% [nap "cpi(m,)"]
1 3 5
6 8 9
% [nap "cpi(m,,)"]
1 3 5
6 8 9
```

10.5.4 `cv(`*main_nao*`[, `*dim_number*`|`*dim_name*`])`

This is simply an alias for `coordinate_variable`.

10.5.5 `derivative(`*a*`[, `*dim_number*`|`*dim_name*`])`

Estimate derivative along specified dimension (default is 0) of array a. The result has the same shape as a.

Example (assuming `vector` has dimension (and coordinate variable) `time`:

```
derivative(vector); # result is derivative with respect to time
```

Examples (assuming `matrix` has dimensions `latitude` and `longitude`):

```
derivative(matrix, 'latitude'); # result is derivative with respect to latitude
derivative(matrix, 0);          # result is derivative with respect to latitude
derivative(matrix);             # result is derivative with respect to latitude
derivative(matrix,'longitude'); # result is derivative with respect to longitude
derivative(matrix, 1);          # result is derivative with respect to longitude
```

The derivative is based on the quadratic through 3 points (provided size of dimension is greater than 2, if only 2 then based on straight line). These points always include the point corresponding

to the result. For interior points, the other 2 are the closest neighbours on each side. For boundary points, these are the 2 closest neighbours.

Let $D(x)$ be the derivative of the quadratic through points (x_0, y_0), (x_1, y_1) and (x_2, y_2).

$$D(x_1) = a_0 y_0 + a_1 y_1 + a_2 y_2$$

where the coefficients a_0, a_1, a_2 are defined by:

$$\begin{aligned} a_0 &= \frac{x_1 - x_2}{(x_1 - x_0)(x_2 - x_0)} \\ a_1 &= \frac{1}{x_1 - x_0} - \frac{1}{x_2 - x_1} \\ a_2 &= \frac{x_1 - x_0}{(x_2 - x_0)(x_2 - x_1)} \end{aligned}$$

10.5.6 `fill_holes`(x, [*max_nloops*])

x is array to be filled.
max_nloops is maximum number of iterations.
Replace missing values by estimates based on means of neighbours. If *max_nloops* is not specified then `fill_holes` continues until there are no missing values.

10.5.7 `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
```

10.5.8 `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
```

10.5.9 `gets_matrix`(*filename*[, *n_header_lines*])

Read text file and return NAO matrix whose rows correspond to the lines in the file. Ignore:

- first *n_header_lines* (default 0) lines
- blank lines
- lines whose first non-whitespace character is '`#`'

Examples:

```
nap "m1 = gets_matrix('matrix1.txt')"
nap "m2 = gets_matrix('matrix2.txt', 3)"; # Skip 1st 3 lines
```

10.5.10 `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.
Examples:

```
% [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
```

10.5.11 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 and 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
```

10.5.12 isMissing(*x*)

1 if *x* missing, 0 if present.
Example:

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

10.5.13 isPresent(*x*)

0 if *x* missing, 1 if present.
Example:

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

10.5.14 magnify_interp(*a*, *mag_factor*)

Magnify each dimension of array *a* by factor defined by the corresponding element of *mag_factor* if this is a vector. If this is a scalar then every dimension is magnified by the same factor. The new values are estimated using multi-linear interpolation.

This function can be used to make images larger or smaller.

Example:

```
% [nap "magnify_interp({{1 2 3}{4 5 6}}, {1 3})"] value
1.00000 1.33333 1.66667 2.00000 2.33333 2.66667 3.00000
4.00000 4.33333 4.66667 5.00000 5.33333 5.66667 6.00000
```

10.5.15 magnify_nearest(*a*, *mag_factor*)

This function is similar to `magnify_interp` except that the new values are defined by the nearest neighbour rather than interpolation.

Example:

```
% [nap "magnify_nearest({{1 2 3}{4 5 6}}, {1 3})"] value
1 1 2 2 2 3 3
4 4 5 5 5 6 6
```

10.5.16 mixed_base(x, b)

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

Following example converts 87 inches to yards, feet and inches:

```
% [nap "mixed_base(87, {3 12})"]
2 1 3
```

10.5.17 nub(x)

Result is vector of distinct values in argument (in same order).

Example:

```
% [nap "nub {{5 -9 5}{1 1 5}}"]
5 -9 1
```

10.5.18 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 and y. The coordinate variables of the result are x and y.

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
```

10.5.19 palette_interpolate(*from*, *to*)

Define a palette by interpolating around the HSV (hue, saturation, value) colour wheel with both S (saturation) and V (value) set to 1. The arguments *from* and *to* are angles in degrees which specify the range of H (hue). Red is 0, green is -240 and blue is 240.

10.5.20 scattered2grid(*xyz*, *ycv*, *xcv*)

Produce a matrix grid from scattered (x, y, z) data using triangulation. Grid points within each triangle are defined by interpolating using a plane through the three vertices of the triangle.

xyz is an $n \times m$ matrix containing data corresponding to n points (x, y, z). The number of columns (m) must be at least 3. Columns 0, 1 and 2 contain x, y and z values respectively. Any further columns are ignored.

ycv and *xcv* specify the coordinate-variables for the grid.

The following example defines a grid from the four points $(2, 2, 0)$, $(6, 4, 0)$, $(2, 4, 4)$ and $(4, 5, 3)$. Note that the missing values in the result correspond to points which are outside of both the triangles produced by the triangulation. You could eliminate these missing values by defining values at all four corners of the grid.

```
% nap "z = scattered2grid({{2 2 0}{6 4 0}{2 4 4}{4 5 3}}, 2 .. 5, 2 .. 6)"
::NAP::1020-1020
% $z
0.00    _    _    _    _
2.00 0.75 0.00    _    _
```

```
4.00 2.50 1.50 0.75 0.00
   _    _ 3.00    _    _
% [nap "z(@2, @2)"]; # Check value at x=2, y=2
0
% [nap "z(@4, @6)"]; # Check value at x=6, y=4
0
% [nap "z(@4, @2)"]; # Check value at x=2, y=4
4
% [nap "z(@5, @4)"]; # Check value at x=4, y=5
3
```

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

Find suitable values for an axis of a graph. These values have a range *within* the interval from *xstart* to *xend*.
xstart: First data value
xend: Final data value
nmax: Maximum 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
```

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

Find suitable values for an axis of a graph. These values have a range which *includes* the interval from *xstart* to *xend*.
xstart: First data value
xend: Final data value
nmax: Maximum 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
```

10.5.23 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
```

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

If $n \geq 0$ 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
```

Chapter 11

Tcl Library Procedures called as Commands

11.1 Introduction

The Nap installation process installs a library of Tcl procedures in directory *tcl_root*`/lib/nap*.*`. This chapter describes procedures which are related to Nap but intended to be called as commands rather than functions.

11.2 Procedures for Formatting Dates and Times

11.2.1 Introduction

Date/time concepts are introduced in section 10.2.1 (Functions for Formatting Dates and Times). The file `date.tcl` defines the two formatting procedures `format_jdn` and `format_mjd`, as well as these functions.

These two formatting procedures use the standard Tcl `clock` command. The range of dates which `clock` can handle depends on the platform. The current (8.4.7) Windows version uses a 32-bit signed integer offset (seconds) from 1970-01-01, allowing dates from 1901-12-13 to 2038-01-19.

The default formats for both procedures are based on the ISO 8601 standard for dates and times. This standard is discussed in many places, including Wikipedia.

11.2.2 Formatting Julian Day Numbers (JDNs)

The command
`format_jdn` *jdn* [*format*]
formats the Julian Day Number *jdn*, one value per line.

jdn can be any Nap expression.

format is that required for the `clock` command. The default format is `"%Y-%m-%d"`, which produces a standard ISO 8601 date in the form '*yyyy-mm-dd*'.

11.2.2.1 Examples

```
% format_jdn 2453305
2004-10-26
% format_jdn 2453305 "%A %B %d, %Y"
Tuesday October 26, 2004
% format_jdn "{0 7} + date2jdn{2004 10 22}"
2004-10-22
2004-10-29
% format_jdn "{0 7} + date2jdn{2004 10 22}" "%y %m %d"
```

```
04 10 22
04 10 29
```

11.2.3 Formatting Modified Julian Dates (MJDs)

The command
`format_mjd` *mjd* [*format*]
formats the Modified Julian Date *mjd*, one value per line.

mjd can be any Nap expression. This is rounded to the nearest second.

format is that required for the `clock` command. The default format is `"%Y-%m-%dT%H:%M:%S"`, which produces a standard ISO 8601 date/time in the form '*yyyy*`-`*mm*`-`*dd*`T`*HH*`:`*MM*`:`*SS*'.

11.2.3.1 Examples

```
% format_mjd 49797.75
1995-03-21T18:00:00
% format_mjd 49797.75 "%I %p on %A %d %B, %Y."
06 PM on Tuesday 21 March, 1995.
% format_mjd "{0 0.5} + dateTime2mjd{2004 10 22 16 30 59}"
2004-10-22T16:30:59
2004-10-23T04:30:59
% format_mjd "{0 0.5} + dateTime2mjd{2004 10 22 16 30 59}" "%y%m%d %H%M"
041022 1630
041023 0430
```

11.3 Binary Input/Output Procedures

11.3.1 Introduction

The following procedures are defined in the file `bin_io.tcl`.

11.3.2 Reading and Writing Simple Binary Files

A simple binary file is a file containing nothing except data of a single data type.

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

Read NAO from binary 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)

11.3.2.2 Procedure `put_nao` [*napExpr* [*fileName*]]

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

11.3.3 Reading and Writing Fortran Unformatted Files

Fortran unformatted files are files consisting of binary records preceded and followed by 32-bit byte-counts.

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

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

11.3.3.2 Procedure `put_bin` *napExpr* [*fileId* [*mode*]]

Write Fortran binary (unformatted) record.
napExpr: Nap expression to be evaluated in caller namespace
fileId: Tcl file handle (default: `stdout`)
mode: `binary` (default) or `swap`

11.3.3.3 Example

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
```

11.3.4 Reading and Writing `cif` Files

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 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)

The main input procedure is `get_cif`, which reads one or more matrices from each of one or more cif files. The main output procedure is `put_cif`, which writes a NAO as an entire cif. These procedures call the low-level procedures `get_cif1` or `put_cif1` for each frame. The default *mode* for `put_cif` and `put_cif1` is `auto` which produces a big-endian file regardless of the platform.

11.3.4.1 Procedure `get_cif` [*options*] *pattern* [*pattern* **...**]

Read one or more matrices from each of one or more cif files (whose names are specified by one or more glob patterns). The result is 2D if only one frame is read, otherwise it is 3D. Check whether byte swapping is needed by examining 1st word in file.
Options:
`-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 a single-frame cif named `7.cif` into a NAO called `in`, then displays it (including the coordinate variables).

```
% nap "in = [get_cif 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
```

11.3.4.2 Procedure `put_cif` *napExpr* [*fileName* [*mode*]]

Write NAO as entire cif.
napExpr: Nap expression to be evaluated in caller namespace
fileName: file name (default: `stdout`)
mode: `auto` (default), `binary` or `swap`

11.3.4.3 Procedure `get_cif1` [*options*] *fileId*

Read next frame from cif (Conmap Input File).
Options:
`-g 0|1`: `1` (default) for geographic mode, `0` for non-geographic mode
`-m` **real**: Input missing value (default: `-7777777.0`)
`-s 0|1`: `0` (default) for binary mode, `1` for swap (byte-swapping) mode
`-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 named `7.cif` into a NAO called `in`, then displays it (including the coordinate variables).

```
% set f [open 7.cif]
file5
% nap "in = [get_cif1 $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
```

11.3.4.4 Procedure `put_cif1` *napExpr* [*fileId* [*mode*]]

Write NAO as frame of cif.
napExpr: Nap expression to be evaluated in caller namespace
fileId: Tcl file handle (default: `stdout`)
mode: `auto` (default), `binary` or `swap`

11.3.5 Writing HDF and netCDF files

11.3.5.1 Procedure `put16` *napExpr fileName variableName*

Write automatically-scaled 16-bit variable to netCDF or HDF file.
napExpr: Nap expression to be evaluated in caller name-space
fileName: name of netCDF (`.nc`) or HDF (`.hdf`) file
variableName: name of 16-bit netCDF variable or HDF SDS
Example:

```
% put16 "{{1.2e4 -99 _}{_ 101 0}}" m.nc matrix
```

11.4 Geographic Procedures

11.4.1 Introduction

The following procedures are defined in the file `geog.tcl`. Note that there are also Geographic Functions.

11.4.2 Write *ARC/INFO GRIDASCII* file using `put_gridascii`

Procedure `put_gridascii` writes a file in ARC/INFO GRIDASCII format. All cells must be squares of the same size. Coordinate variable 0 (usually latitude) can be either ascending or descending.

11.4.2.1 Usage

`put_gridascii` *expr filename* [*missing_value_string*]

expr Nap expression defining a matrix with coordinate variables (normally latitude and longitude)

filename Pathname of the output file

missing_value_string String representing missing value in file (Default: `1e9`)

11.4.2.2 Example

```
% put_gridascii mat abc.txt -999
```

11.4.3 Write *Surfer* file using `put_text_surfer`

Procedure `put_text_surfer` writes a file in *Surfer* format. Surfer is a package developed by *Golden Software*.

11.4.3.1 Usage

`put_text_surfer` *expr* [*filename*] [*format*]

expr Nap expression defining a matrix with coordinate variables

filename Pathname of the output file. If none then writes to standard output.

format C format specification of each output element (Default: `"%g"`)

11.4.3.2 Example

```
% put_text_surfer mat abc.txt %8.4f
```

11.5 Map Projection Procedures

11.5.1 Introduction

The following procedures are defined in the file `projection.tcl`. Note that these procedures are now deprecated in favour of the PROJ.4 functions described in section 5.5.11.

11.5.2 `projection` *code* p_0 p_1 p_2 ...

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

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

- `CylindricalEquidistant`: p_0 = x-origin (default: `""`)
- `Mercator`: p_0 = x-origin (default: `""`)
- `NorthPolarEquidistant`: North Polar azimuthal equidistant
- `SouthPolarEquidistant`: South Polar azimuthal equidistant
- `SouthPolarStereographic`: As used by IASOS

p_0 p_1 p_2 ... define parameters of the projection. Some projections use p_0 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.

11.6 Miscellaneous Procedures

11.6.1 Procedure `size_of` *dataType*

Number of bytes in *dataType*.

This procedure is defined in the file `bin_io.tcl`.

Example:

```
% size_of i16
2
```

Chapter 12

Tk GUI Procedures

12.1 Introduction

The Nap installation process installs a library of Tcl procedures in directory *tcl_root*`/lib/nap*.*`. This chapter describes procedures which define GUIs related to Nap.

12.2 Caps/Nap Menu

12.2.1 Introduction

The distributed `wish` startup file '`wishrc.tcl`' (which is copied to file '`.wishrc`' on unix systems) ends with the command '`caps_nap_menu`', which displays a GUI menu consisting of three buttons labelled *Browse*, *Command* and *Help*. This command '`caps_nap_menu`' is defined by the file '`caps_nap_menu.tcl`'. Each of these three buttons is used to display a sub-menu detailed in the following sections.

12.2.2 Browse

The *Browse* menu provides access to browsers for Tcl variables and a variety of files. These file browsers typically provide facilities to:

- Select a file using the `choose_file` GUI (see section 12.3).
- Select data from this file.
- Display this data as text.
- Plot this data (as an XY graph or a 2D image) using `plot_nao` (see section 12.4).
- Print this data.
- Convert this data to a NAO.

Detailed help is available within each browser.

12.2.2.1 Tcl Variables

This facility for browsing *Tcl variables* is defined by the file `browse_var.tcl`. It can be used to display the names and values of all tcl variables (including arrays), but has particular facilities for those referencing NAOs. The menu buttons have the following functions:

namespace Use tree widget to set the namespace.

list List tcl variables in the namespace matching the glob pattern. Display a line for each matching tcl variable. This line contains the variable's name and value provided there is room. Otherwise the line is truncated. You can click on a line to display:

- full value of an ordinary tcl variable or array whose line is truncated as above
- a NAO as either text or graph/image as specified by the radio-button

help Display *Help on Tcl Variable Browser*.

cancel Remove *Tcl Variable Browser* widget.

12.2.2.2 *AVHRR* and *ATSR* Satellite Files

This item is only available if the Caps package is loaded.

12.2.2.3 *CIF* Files

CIF files originated at Melbourne University. Their use has been largely replaced by netCDF but some data still exists in this format. This facility for browsing *CIF* files is defined by the file `browse_cif.tcl`.

12.2.2.4 *HDF* and *netCDF* Files

12.2.2.4.1 Introduction

HDF and netCDF are two similar common file formats used for data in the form of arrays and grids, especially in the earth sciences such as Meteorology and Oceanography. See section 2.5 for further information about HDF and netCDF.

The term *variable/SDS* is used below. This means *variable* for netCDF and *SDS* for HDF.

The HDF/netCDF browser is a powerful tool which allows one to select a variable/SDS or attribute from a displayed tree, then (for variable/SDS) select a region within it.

The HDF/netCDF browser is defined by the file `hdf.tcl`.

12.2.2.4.2 Instructions

1. Use the `choose-file` GUI to open an input file. Instructions can be displayed by pressing this GUI's own `help` button. Opening the file should result in the display of a *file structure tree*.

2. Use this tree as follows to select either a variable/SDS or an attribute. (The default selection is a variable/SDS with the maximum number of elements.)
   - Click on a variable/SDS to select it and display the spatial sampling widget.
   - Click on a '`+`' to display attribute names.
   - Click on an attribute to select it and display its value.

3. The spatial sampling widget allows you to select part of a variable/SDS. (The entire variable/SDS is selected by default.)
   Each dimension is represented by a row containing one or two lines. The first line represents subscript values. If a coordinate variable exists then it is represented on a second line.
   Change a subscript using any of the following:
   - Drag the slider along the scale widget. This is convenient for coarse adjustment.
   - Click on the spinbox arrows or scale troughs.
   - Press the keyboard up/down keys.
   - Use the keyboard to enter numbers. Fractional subscript values can be used to produce magnification.
   - On an image, drag the mouse to define a bounding box.
   - Press the `Dimension` button to restore all defaults.
   - Press the `From`, `To` or `Step` column heading button to restore defaults in a column.
   - Press the row heading buttons to toggle a row between defaults and saved values.

   The values selected along a dimension are defined as follows:
   - If $step > 0$ then *from*, *to* and *step* define an arithmetic progression.
   - If $step = 0$ and expression is blank then use single value *from*.
   - If $step = 0$ and expression is not blank then use this expression.

4. The following buttons along the bottom are used to select an action:
   `Range`: Display minimum and maximum value.
   `Text`: Display start of data as text.
   `Graph`: Use `plot_nao` to display data as XY graph(s).

`Image`: Use `plot_nao` to display data as 2D image(s).
`Animate`: Animate window-sequence produced by `Graph` or `Image`.
`NAO`: Create Numeric Array Object.
`Re-read`: Force a read (e.g. after rewriting the file).

Select `Raw` mode if you want the following attributes to be ignored:
`scale_factor, add_offset, valid_min, valid_max, valid_range`.

12.2.2.5 *Image* Files

This facility for *viewing image files* is defined by the file `vif.tcl`. Standard Tk supports the PPM, PGM and GIF formats. If the package `Img` is installed then it will be used, giving support for BMP, XBM, XPM, PNG, JPEG and TIFF as well. The ActiveTcl distribution of Tcl/Tk includes `Img`.

12.2.3 Command

The *Command* menu allows the following common Tcl commands to be executed with a mouse click:

`history` Display command history. This differs from the command-line `history` command in that it uses a scrolled window. Note that `tkcon` also provides:
- *History* entry in main menu
- *History* entry within *File/Save*

`exit` Quit.

12.2.4 Help

The *Help* menu provides access to local and Web documentation on:
- Caps/Nap Menu
- Tcl/Tk
- Nap
- Caps

Note that the Web versions may be more recent than the installed local documentation.

12.3 choose_file GUI

12.3.1 Introduction

The `choose_file` GUI is used to select and open input files. It contains the following three lines:
- *Filename* button and spinbox widget
- *Directory* button and entry widget
- *Glob Filter* entry widget

If you are already in the desired directory then the spinbox may be all you need. The *file selection dialog* GUI can be used to select both the directory and the file. The *directory (folder) selection dialog* GUI is useful when the directory contains many files matching the filter.

The `choose_file` GUI is defined by the file `choose_file.tcl`.

12.3.2 Instructions

1. If any field is too narrow then resize the window by dragging its edge.
2. Select an input file using any of the following:
   - Type into any of the three entry fields.
   - Press the `Filename` button to use the *file selection dialog* GUI.
   - Press the `Directory` button to use the *directory (folder) selection dialog* GUI.

- Click on the spinbox arrows (or press the keyboard up/down keys) to spin through the files matching the filter.

3. Press the `Open` button to open the file. This can also be done by pressing the `Enter` (a.k.a. `Return`) key in the `Filename` entry field.

12.3.3 Usage

The following `choose_file` command creates a new temporary GUI window, accepts input from the user and then returns (as the result) the pathname of the specified input file:
`choose_file` ?*parent*? ?*filter*? ?*geometry*?
parent: Parent window (default: `"."`)
filter: Initial glob filter (default: `"*"`)
geometry: Value as follows (default: `"NW"`):
`""` : Pack in parent (If parent is `""` then create toplevel anywhere)
`"NE"` : North-west corner of toplevel at north-east corner of parent (cannot be `""`)
`"NW"` : North-west corner of toplevel at north-west corner of parent (cannot be `""`)
`"SE"` : North-west corner of toplevel at south-east corner of parent (cannot be `""`)
`"SW"` : North-west corner of toplevel at south-west corner of parent (cannot be `""`)
other: Normal Tk geometry string. If parent is `""` then pack in it, else ignore parent.

12.4 Visualisation using procedure `plot_nao`

12.4.1 Introduction

A NAO can be visualised using procedure `plot_nao`, which can represent it by:
- xy graphs (1D and 2D NAOs)
- bar-charts and histograms (1D and 2D NAOs)
- color-coded z-images and maps (2D NAOs)
- RGB z-images and maps (3D NAOs with three layers for red, green and blue)
- tiled images (multiple color-coded z-images on a page) (3D NAOs)

If there are additional dimensions then it is possible to generate multiple frames which can be animated. See examples.

The right mouse button displays a menu. The left mouse button saves (x,y,z) values (z-images only) which can be written to a file.

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

12.4.2 Usage

`plot_nao` *expression ?options?*

12.4.2.1 Options

Most options can be set using either the above *command-line* or the *options menu*. Command-line options are as follows:
`-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 xy graphs or bars. (Default: `black red green blue yellow orange purple grey aquamarine beige`)
`-columns` *int*: (tiled-plot only) number of columns of tiles on page
`-dash` *list*: Dash patterns of xy-graph lines. (Default: `""` i.e. all full lines) Each element is `""` for full line, `" "` for no line, or standard Tk dash pattern (See entry for *canvas* in Tk manual).
`-discrete` 0 or 1: 1 = Discrete colors between major z tick marks. (Default: 0)

`-filename` *name with extension* `.ps, .gif, .jpeg,` *etc.:* File produced by `-print` (Default: Print rather than writing a file)
`-fill` 0 or 1: 1 = Scale PostScript to fill page. (Default: 0)
`-font_standard` *font*: Main font. (Default: `"courier 10"`)
`-font_title` *font*: Font for title. (Default: `"courier 16"`)
`-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. (E.g. `"-geometry +0+0"` for top left corner)
`-height` *int*: Desired height (screen units). Not used for tiled-plot.
- Type `xy/bar`: Height of whole window (Default: automatic)
- Type `z`: Image height (can be '*min max*' for range) (Default: NAO dim if within limits)

`-key` *int*: width (pixels) of key. No key if 0 or blank. (Default: 30)
`-labels` *list*: Labels of tiles or xy-graphs
`-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)
`-oversize_prompt` 0 or 1: 1 = prompt if image is larger than screen. (Default: 1)
`-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 and 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 print/write (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*: Symbol drawn at each point of xy-graph. Can be `plus, square, circle, cross, splus, scross, triangle` or single character (e.g. `"*"`) (Default: `""` i.e. none)
`-title` *string*: title (Default: NAO label (if any) else *expression*
`-type` *string* plot-type (`bar, tile, 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 default type is `"z"`
`-width` *int*: Desired width (screen units). Not used for tiled-plot.
- Type `xy/bar`: Width of whole window (Default: automatic)
- Type `z`: Image width (can be '*min max*' for range) (Default: NAO dim if within limits)

`-xaxis` 0 or 1: Draw x-axis? 0 = no, 1 = yes. (Default: 1)
`-xflip` 0 or 1: Flip left-right? 0 = no, 1 = yes. (Default: 0)
`-xlabel` *string*: x-axis label (Default: name of last (final) dimension)
`-xproc` *string*: name of procedure to format x-axis tick values (Default: none)
`-xticks` *expression*: Major tick marks of x-axis (Default: automatic)
`-yaxis` 0 or 1: Draw y-axis? 0 = no, 1 = yes. (Default: 1)
`-yflip 0, 1, ascending or geog`: Flip upside down? (Default: `geog`)
0 = no,
1 = yes,
`ascending` = 'if y ascending',
`geog` = 'if ascending and (y_dim_name = `latitude` or y_unit = `degrees_north` (or equiva-

lent))'
`-ylabel` *string*: y-axis label (Default: name of 2nd-last dimension)
`-yproc` *string*: name of procedure to format y-axis tick values (Default: none)
`-yticks` *expression*: Major tick marks of y-axis (Default: automatic)
`-zlabels` *list*: z-axis labels of values 0, 1, 2, ... (Default: none)
`-zticks` *expression*: Major tick marks of z-axis (Default: automatic)

12.4.3 Examples

You can cut and paste the following examples into tkcon or wish.

12.4.3.1 x-y graphs and bar-charts

```
nap "sales = {
    {2 5 1 3 5 7 9 1 2 9 1 0}
    {9 2 5 5 3 9 2 0 8 8 3 8}
}"
nap "month = 1 .. 12"
$sales set coord "" month
$sales set label "Car sales"
plot_nao sales -labels "Joe Mary" -xtick "1..12" -dash {{} .} -symbols "plus cross"
proc format_x x {lindex {{} Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec} $x}
plot_nao sales -labels "Joe Mary" -xtick "1..12" -type bar -xproc format_x
```

12.4.3.2 Scattergram

```
nap "x = {1.1 3.2 2.0 5.9 7.7 4.5 6.3}"
nap "y = {5.0 4.1 9.9 3.7 1.2 2.1 4.5}"
$y set coo x
plot_nao y -symbols plus -dash " "
```

12.4.3.3 Color-coded z-images

These examples define and use the 2D NAO `z`.

```
nap "n = 200"
nap "x = n ... 0.0 .. 10.0"
nap "y = x(-)"
nap "z = sin(x) * sin(reshape(n#y, 2#n))"
$z set coo y x
plot_nao z
plot_nao z -zticks "-1 .. 1 ... 0.2" -discrete 1; # discrete colors
plot_nao "nint(z+1)" -zlabels {{zero (0)} {one (1)} {two}}; # labelled values 0, 1, 2
```

12.4.3.4 RGB z-images, tiles and animation

These examples define and use the 3D NAO `z3d`. This is defined using `z` defined above.

```
nap "z3d = z /// z*z // z**3 // z**4"
plot_nao z3d; # layer 0 = red, layer 1 = green, layer 2 = blue, layer 3 is ignored
plot_nao z3d -type tile -labels {z {z * z} z**3 z**4} -title "powers of z"
set frames [plot_nao z3d -rank 2]; # create four 2D frames
animate $frames; # animate these frames
```

12.4.3.5 Printing and writing files

Note that the `-print 1` option can be used to print and write automatically in batch mode operation. It may be necessary to specify '`-geometry +0+0`' to ensure the window is entirely visible.

```
nap "x = 200 ... 0 .. 4p"
nap "y = sin x"
$y set coo x
plot_nao y -print 1; # Print on default printer
plot_nao y -print 1 -geometry +0+0 -filename sin.ps; # write postscript file sin.ps
plot_nao y -print 1 -geometry +0+0 -filename sin.jpeg; # write JPEG image file sin.jpeg
```

Chapter 13

Nap Internal Details

13.1 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
`f64`	64-bit floating-point
`ragged`	compressed
`boxed`	slot numbers (used as pointers to NAOs)

See section 4.4.1 (Data Types) for further details of these types.

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.

format Text containing C format for printing.

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.

13.2 Nap Photo Image Format

Nap defines a new photo image format. This enables one to use Tk's '`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)
```

Chapter 14

Demonstrations of Nap

14.1 Introduction to Demonstrations of Nap

These are logs of demonstrations given during courses on Nap. There is some overlap with material in previous sections.

14.2 Simple Nap

The following calculates areas of circles.

```
% set r 10
10
% nap "area = 1p * r ** 2"; # 1p = pi
::NAP::2125-2125
% $area
314.159
% set area
::NAP::2125-2125
% ::NAP::2125-2125
314.159
% [nap "1p * r ** 2"]
314.159
% [nap "1p * r ** 2"] all
::NAP::2141-2141  f64  MissingValue: NaN  References: 0
Value:
314.159
% nap "r = 1 .. 3 ... 0.5"; # arithmetic progression from 1 to 3 in steps of 0.5
::NAP::2150-2150
% $r
1 1.5 2 2.5 3
% nap "area = 1p * r ** 2"
::NAP::2160-2160
% $area
3.14159 7.06858 12.5664 19.635 28.2743
% $area set coord r
% $area all
::NAP::2160-2160  f64  MissingValue: NaN  References: 1
Dimension 0   Size: 5      Name: r         Coordinate-variable: ::NAP::2150-2150
Value:
3.14159 7.06858 12.5664 19.635 28.2743
% plot_nao area
.plot_nao1
```

```
% [nap "r /// area"]; # /// joins over new dimension
 1.00000  1.50000  2.00000  2.50000  3.00000
 3.14159  7.06858 12.56637 19.63495 28.27433
% [nap "transpose(r /// area)"]
 1.00000  3.14159
 1.50000  7.06858
 2.00000 12.56637
 2.50000 19.63495
 3.00000 28.27433
```

14.3 Constants

14.3.1 Scalar Constants

```
% [nap "14"] all
::NAP::14465-9991  i32  MissingValue: -2147483648  References: 0
Value:
14
% [nap "i16(14)"] all; # i16 = 16-bit signed integer
::NAP::14467-9991  i16  MissingValue: -32768  References: 0
Value:
14
% [nap "14i16"] all
::NAP::14469-9991  i16  MissingValue: -32768  References: 0
Value:
14
% [nap "14u8"] all; # u8 = unsigned 8-bit integer
::NAP::14471-9991  u8  MissingValue: (NULL)  References: 0
Value:
14
% [nap "4.0"] all
::NAP::14473-9991  f64  MissingValue: NaN  References: 0
Value:
4
% [nap "4."] all
4.
 ^
parse error, unexpected '.', expecting $
Error at line 791 of file /usr/share/bison/bison.simple
% [nap ".4"] all
.4
^
parse error, unexpected '.'
Error at line 791 of file /usr/share/bison/bison.simple
% [nap "4f32"] all
::NAP::14475-109  f32  MissingValue: NaN  References: 0
Value:
4
% [nap "2r3"] all
::NAP::14477-109  f64  MissingValue: NaN  References: 0
Value:
0.666667
% [nap "2r3f32"] all
::NAP::14478-109  f32  MissingValue: NaN  References: 0
Value:
```

```
0.666667
% [nap "1e4"] all
::NAP::14480-109  f64  MissingValue: NaN  References: 0
Value:
10000
% [nap "1p1"] all
::NAP::14481-109  f64  MissingValue: NaN  References: 0
Value:
3.14159
% [nap "1p"]
3.14159
% [nap "180p-1f32"] all
::NAP::14482-109  f32  MissingValue: NaN  References: 0
Value:
57.2958
% [nap "1r3p1"] all
::NAP::14484-109  f64  MissingValue: NaN  References: 0
Value:
1.0472
% [nap "1i"] all; # infinity
::NAP::22-22  f64  MissingValue: NaN  References: 0
Value:
Inf
% [nap "-1i"] all; # -infinity
::NAP::24-24  f64  MissingValue: NaN  References: 0
Value:
-Inf
% [nap "1n"] all; # NaN
::NAP::25-25  f64  MissingValue: NaN  References: 0
Value:
_
% [nap "_"] all; # missing
::NAP::26-26  i32  MissingValue: -2147483648  References: 0
Value:
-
```

14.3.2 Array Constants

```
% [nap "{3 -5 _ 9}"] all
::NAP::14486-9829  i32  MissingValue: -2147483648  References: 0
Dimension 0   Size: 4      Name: (NULL)    Coordinate-variable: (NULL)
Value:
3 -5 _ 9
% [nap "{3 3#-5 _ 9}"] all
::NAP::14489-109  i32  MissingValue: -2147483648  References: 0
Dimension 0   Size: 6      Name: (NULL)    Coordinate-variable: (NULL)
Value:
3 -5 -5 -5 _ 9
% [nap "{{1 3 5}{2 4 6}}"] all
::NAP::14492-9829  i32  MissingValue: -2147483648  References: 0
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
% [nap "{
```

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

14.3.3 String Constants

```
% [nap "'Hello world'"]
Hello world
% [nap "`can't`"]
can't
```

14.4 Arithmetic

14.4.1 Binary arithmetic operators

```
% [nap "2+2"]
4
% [nap "{4.2 9} + {5 -1}"]
9.2 8
% [nap "{4.2 9} + 2"]
6.2 11
% [nap "{4.2 9} - 2"]
2.2 7
% [nap "{4.2 9} * 2"]
8.4 18
% [nap "{4.2 9} / 2"]
2.1 4.5
% [nap "{4.2 9} % 2"]; # remainder (mod)
0.2 1
% [nap "{4.2 9} ** 2"]; # power
17.64 81
% [nap "{4 5 8} < 5"]
1 0 0
% [nap "{4 5 8} > 5"]
0 0 1
% [nap "{4 5 8} <= 5"]
1 1 0
% [nap "{4 5 8} >= 5"]
0 1 1
% [nap "{4 5 8} == 5"]; # equal
0 1 0
% [nap "{4 5 8} != 5"]; # not equal
1 0 1
% [nap "{0 0 1 1} && {0 1 0 1}"]; # and
0 0 0 1
```

```
% [nap "{0 0 1 1} || {0 1 0 1}"]; # or
0 1 1 1
% [nap "{4 5 8} <<< 5"];# lesser of (min)
4 5 5
% [nap "{4 5 8} >>> 5"];# greater of (max)
5 5 8
```

14.4.2 Unary arithmetic operators - +

```
% [nap "- {6 -3}"]
-6 3
% nap "x = 7"
::NAP::14583-9829
% $x all
::NAP::14583-9829  i32  MissingValue: -2147483648  References: 1  Unit: (NULL)
Value:
7
% nap "y = x"
::NAP::14583-9829
% $y all
::NAP::14583-9829  i32  MissingValue: -2147483648  References: 2  Unit: (NULL)
Value:
7
% $x all
::NAP::14583-9829  i32  MissingValue: -2147483648  References: 2  Unit: (NULL)
Value:
7
% nap "y = +x"; # new copy of
::NAP::14587-109
% $y all
::NAP::14587-109  i32  MissingValue: -2147483648  References: 1  Unit: (NULL)
Value:
7
% $x all
::NAP::14583-9829  i32  MissingValue: -2147483648  References: 1  Unit: (NULL)
Value:
7
```

14.4.3 Ternary choice operator ?:

```
% expr 1 ? 9 : 3
9
% expr 0 ? 9 : 3
3
% nap "x = -3 .. 3"
::NAP::14595-9985
% $x value
-3 -2 -1 0 1 2 3
% [nap "x < 0 ? -9 : x"] value
-9 -9 -9 0 1 2 3
% [nap "x %3 == 0 ? _ : x"] value
_ -2 -1 _ 1 2 _
% [nap "x < 0 ? -1 : x > 0 ? 1 : 0"] value
-1 -1 -1 0 1 1 1
```

14.5 OOCs (Object-Oriented Commands)

14.5.1 Display contents (data and attributes) of NAO

NAO stands for *n-dimensional array object*.

14.5.1.1 Default OOC

```
% [nap "2 ** (0 .. 8)"]
1 2 4 8 16 32 ..
% [nap "2 ** (0 .. 8)"] -columns 8 -format %.1f
1.0 2.0 4.0 8.0 16.0 32.0 64.0 128.0 ..
% [nap "2 ** (0 .. 8)"] -columns -1; # -1 = infinity
1 2 4 8 16 32 64 128 256
```

14.5.1.2 Method `value`: Display all lines and columns

```
% [nap "2 ** (0 .. 8)"] value
1 2 4 8 16 32 64 128 256
% [nap "2 ** (0 .. 8)"] v; # can abbreviate, provided unique
1 2 4 8 16 32 64 128 256
```

14.5.1.3 Method `head`

```
% [nap "2 ** (0 .. 8)"] head
::NAP::233-233  f32  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 9      Name: (NULL)    Coordinate-variable: (NULL)
```

14.5.1.4 Method `all`

```
% [nap "2 ** (0 .. 8)"] all
::NAP::242-242  f32  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 9      Name: (NULL)    Coordinate-variable: (NULL)
Value:
1 2 4 8 16 32 ..
% [nap "2 ** (0 .. 8)"] all -columns -1 -lines -1; # all columns and lines
::NAP::252-252  f32  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 9      Name: (NULL)    Coordinate-variable: (NULL)
Value:
1 2 4 8 16 32 64 128 256
```

14.5.1.5 Individual attributes

```
% [nap "x = {{0 2.4 1}{3.6 2 -9}}"]
 0.0  2.4  1.0
 3.6  2.0 -9.0
% $x datatype
f64
% $x shape
2 3
% $x missing
NaN
```

14.5.2 Method `set`: Change these contents

14.5.2.1 Set *missing value*

```
% [nap "sum(x)"]
```

```
3.6 4.4 -8
% $x set missing -9
% $x missing
-9
% $x
0.0 2.4 1.0
3.6 2.0   _
% [nap "sum(x)"]
3.6 4.4 1
% $x set value "{-1 -3}" "1,{0 2}"; # Set x(1,0) = -1, x(1,2) = -3
% $x
 0.0  2.4  1.0
-1.0  2.0 -3.0
```

14.5.2.2 Set *coordinate variable*

```
% nap "x = 999 ... -2p .. 2p"
::NAP::339-339
% $x all
::NAP::339-339  f64  MissingValue: NaN  References: 1  Unit: (NULL)
Dimension 0   Size: 999    Name: (NULL)    Coordinate-variable: (NULL)
Value:
-6.28319 -6.27059 -6.258 -6.24541 -6.23282 -6.22023 ..
% nap "y = sin x"
::NAP::341-341
% $y all
::NAP::341-341  f64  MissingValue: NaN  References: 1  Unit: (NULL)
Dimension 0   Size: 999    Name: (NULL)    Coordinate-variable: (NULL)
Value:
2.44921e-16 0.0125912 0.0251804 0.0377657 0.0503449 0.0629162 ..
% $y set coord x
% $y all
::NAP::341-341  f64  MissingValue: NaN  References: 1  Unit: (NULL)
Dimension 0   Size: 999    Name: x         Coordinate-variable: ::NAP::339-339
Value:
2.44921e-16 0.0125912 0.0251804 0.0377657 0.0503449 0.0629162 ..
% plot_nao y; # Display graph. Note that x axis corresponds to coordinate variable 'x'
.win0
```

14.5.3 Write data to (netCDF) file

```
% $y netcdf test.nc sin; # write to variable "sin" in netCDF file "test.nc"
```

14.6 Built-in Functions

14.6.1 Built-in elemental functions

An *elemental function* is one whose result

- has the same shape as its argument
- consists of elements defined by applying the function to the corresponding elements of the argument

14.6.1.1 Mathematical Elemental Functions

```
% [nap "sqrt(5)"]
2.23607
```

```
% nap "x = {1p 0 0.5p -1r6p}"
::NAP::79-79
% $x all
::NAP::79-79  f64  MissingValue: NaN  References: 1  Unit: (NULL)
Dimension 0   Size: 4      Name: (NULL)    Coordinate-variable: (NULL)
Value:
3.14159 0 1.5708 -0.523599
% [nap "180p-1 * x"]; # in degrees
180 0 90 -30
% [nap "sin(x)"]
1.22461e-16 0 1 -0.5
% [nap "sin x"]; # parentheses not needed!
1.22461e-16 0 1 -0.5
% [nap "abs(x)"]
3.14159 0 1.5708 0.523599
% [nap "sign(x)"]
1 0 1 -1
% [nap "nint(x)"] all; # nearest integer
::NAP::91-91  f64  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 4      Name: (NULL)    Coordinate-variable: (NULL)
Value:
3 0 2 -1
% [nap "floor(x)"] all; # nearest integer <= x
::NAP::93-93  f64  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 4      Name: (NULL)    Coordinate-variable: (NULL)
Value:
3 0 1 -1
% [nap "ceil(x)"] all; # nearest integer >= x
::NAP::95-95  f64  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 4      Name: (NULL)    Coordinate-variable: (NULL)
Value:
4 0 2 0
```

14.6.1.2 Type-conversion Elemental Functions

```
% [nap "f32(x)"] all
::NAP::97-97  f32  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 4      Name: (NULL)    Coordinate-variable: (NULL)
Value:
3.14159 0 1.5708 -0.523599
% [nap "i16(x)"] all
::NAP::100-100  i16  MissingValue: -32768  References: 0  Unit: (NULL)
Dimension 0   Size: 4      Name: (NULL)    Coordinate-variable: (NULL)
Value:
3 0 1 0
% [nap "u16(x)"] all
::NAP::103-103  u16  MissingValue: 65535  References: 0  Unit: (NULL)
Dimension 0   Size: 4      Name: (NULL)    Coordinate-variable: (NULL)
Value:
3 0 1 _
% [nap "i32('ABCXYZ')"]
65 66 67 88 89 90
% [nap "c8({65 66 67 88 89 90})"] all
::NAP::194-194  c8  MissingValue: (NULL)  References: 0  Unit: (NULL)
Dimension 0   Size: 6      Name: (NULL)    Coordinate-variable: (NULL)
Value:
```

```
ABCXYZ
% [nap "c8{65 66 67 88 89 90}"]; # parentheses not needed
ABCXYZ
```

14.6.1.3 Elemental Functions which test for Special Values

```
% [nap "x = {2.5 0 -9 _ 7}"] all
::NAP::201-201  f64  MissingValue: NaN  References: 1  Unit: (NULL)
Dimension 0   Size: 5      Name: (NULL)    Coordinate-variable: (NULL)
Value:
2.5 0 -9 _ 7
% $x set missing -9
% $x all
::NAP::201-201  f64  MissingValue: -9  References: 1  Unit: (NULL)
Dimension 0   Size: 5      Name: (NULL)    Coordinate-variable: (NULL)
Value:
2.5 0 _ _ 7
% [nap "isPresent(x)"]
1 1 0 0 1
% [nap "isMissing(x)"]
0 0 1 1 0
% [nap "isnan(x)"]
0 0 0 1 0
```

14.6.1.4 Elemental Function `random`, which generates random numbers

```
% [nap "random(100)"]; # random real number from 0 to 100
62.8871
% [nap "1 + floor(random(20 # 6))"] value; # 20 throws of dice
3 4 6 6 4 5 1 4 1 2 1 5 1 3 1 1 6 2 4 6
```

14.6.2 Non-elemental functions

14.6.2.1 Meta-data functions

```
% nap "matrix = {
{2 4 0}
{9 1 7}
}"
::NAP::253-253
% $matrix all
::NAP::253-253  i32  MissingValue: -2147483648  References: 1  Unit: (NULL)
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
2 4 0
9 1 7
% [nap "shape(matrix)"]
2 3
% [nap "rank(matrix)"]
2
% [nap "shape(shape(matrix))"]; # same as this
2
% [nap "nels(matrix)"]; # number of elements
6
% [nap "prod(shape(matrix))"]; # same as this
6
```

14.6.2.2 Reduction functions applied to vector

```
% nap "x = f32{3 7 _ 1}"
::NAP::278-278
% $x a; # can abbreviate 'all' to 'a'
::NAP::278-278  f32  MissingValue: NaN  References: 1  Unit: (NULL)
Dimension 0   Size: 4      Name: (NULL)    Coordinate-variable: (NULL)
Value:
3 7 _ 1
% [nap "sum(x)"]
11
% [nap "prod(x)"]; # product
21
% [nap "min(x)"]
1
% [nap "max(x)"]
7
% [nap "count(x)"]; # number not missing
3
% [nap "sum(x) / count(x)"]; # mean
3.66667
```

14.6.2.3 Reduction functions applied to `matrix` defined above

```
% [nap "sum(matrix)"]; # sum of each column
11 5 7
% [nap "sum(matrix) / count(matrix)"]; # mean of each column
5.5 2.5 3.5
% [nap "sum(matrix, 1)"]; # sum of each row
6 17
% [nap "sum(matrix,1) / count(matrix,1)"]; # mean of each row
2 5.66667
```

14.6.2.4 Partial sum function `psum`

```
% [nap "psum(x)"]
3 10 10 11
```

14.7 Constructing Arrays

14.7.1 Arithmetic Progression

```
% [nap "3 .. 8"]l # Default step is either 1
3 4 5 6 7 8
% [nap "8 .. 3"]; # or -1
8 7 6 5 4 3
% [nap "-1 .. 2 ... 1r2"] value; # From -1 to 2 in steps of 1/2
-1 -0.5 0 0.5 1 1.5 2
% [nap "-1 .. 0.9 ... 1r2"] value; # Final step may be smaller!
-1 -0.5 0 0.5 0.9
% [nap "4 ... 3 .. 8"]; # 4-element vector from 3 to 8
3 4.66667 6.33333 8
```

14.7.2 Catenate: ('//')

```
% [nap "{3 1 9} // {0 -1 5}"]
3 1 9 0 -1 5
```

14.7.3 Laminate (‘///’)

```
% [nap "{3 1 9} // {0 -1 5}"]
% [nap "{3 1 9} /// {0 -1 5}"]
 3  1  9
 0 -1  5
```

14.7.4 Reverse (niladic ‘-’)

```
% nap "x = {9 1 0 2 3}"
::NAP::24-24
% [nap "x(-)"]
3 2 0 1 9
```

14.7.5 Replicate (binary ‘#’)

```
% [nap "{2 0 1 1 0} # x"]
9 9 0 2
% [nap "2 # x"] value
9 9 1 1 0 0 2 2 3 3
% [nap "(x % 2 == 0) # x"]; # select even elements
0 2
% [nap "(x % 2 == 0)"]
0 0 1 1 0
```

14.7.6 Function `reshape`

```
% nap "m = reshape(1 .. 6, {3 4})"
::NAP::126-126
% $m
1 2 3 4
5 6 1 2
3 4 5 6
% [nap "reshape(m)"] value
1 2 3 4 5 6 1 2 3 4 5 6
```

14.7.7 Function `transpose`

```
% [nap "transpose(m)"] value
1 5 3
2 6 4
3 1 5
4 2 6
```

14.7.8 Function `sort`

```
% [nap "sort({3.6 -1 0 -9 9})"]
-9 -1 0 3.6 9
```

14.8 Indexing

14.8.1 Indexing of vectors

```
% nap "temperature = {20 21 20.6}"; # temperatures at time 1000, 1200, 1400
::NAP::134-134
% nap "time = {10 12 14}"
::NAP::136-136
```

```
% $temperature set coo time
% $temperature a
::NAP::134-134  f64  MissingValue: NaN  References: 1
Dimension 0   Size: 3      Name: time      Coordinate-variable: ::NAP::136-136
Value:
20 21 20.6
% [nap "temperature(0)"]
20
% [nap "temperature(1)"]
21
% [nap "temperature(2)"]
20.6
% [nap "temperature(3)"]
20
% [nap "temperature(-1)"]
20.6
% [nap "temperature(-2)"]
21
% [nap "temperature(0.5)"]
20.5
% [nap "temperature(1.5)"]
20.8
% [nap "temperature({1 0 2 -1})"]
21 20 20.6 20.6
% [nap "temperature{1 0 2 -1}"]
21 20 20.6 20.6
% [nap "temperature(0 .. 2 ... 0.5)"]
20 20.5 21 20.8 20.6
```

14.8.2 Inverse indexing of vectors (position of specified value)

```
% [nap "{2 5 7 2 5} @@@ {7 2 4 5}"]; # @@@ is 'match' (1st to match exactly)
2 0 _ 1
% $time
10 12 14
% [nap "time @ 14"]; # Find position (index) of value 14 in time
2
% [nap "time @ 11"]; # @ is 'interpolate'
0.5
% [nap "time @@ 13.1"]; # @@ is 'nearest'
2
% [nap "time @ {14 11 10 11.5}"]
2 0.5 0 0.75
```

14.8.3 Indirect indexing (via coordinate variable)

```
% [nap "temperature(time @ 14)"]
20.6
% [nap "temperature(     @ 14)"]; # Left operand of @ defaults to coordinate variable
20.6
% [nap "temperature(@ 11)"]
20.5
% [nap "temperature(time @@ 13.1)"]
20.6
% [nap "temperature(@@ 13.1)"]
20.6
```

```
% [nap "temperature(time@{14 11 10 11.5})"]
20.6 20.5 20 20.75
% [nap "temperature(@{14 11 10 11.5})"]
20.6 20.5 20 20.75
```

14.8.4 Shape-preserving indexing

```
% [nap "temperature(0.5)"] a; # (a = all) Both index and result are scalars
::NAP::51955-9298  f32  MissingValue: NaN  References: 0
Value:
20.5
% [nap "temperature({0.5 0 -1})"] a; # Both index and result are 3-element vectors
::NAP::51961-9977  f32  MissingValue: NaN  References: 0
Dimension 0   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
20.5 20 20.6
% [nap "temperature({{0 0.5 -1}{1 0 3}})"] a; # Both index and result are 2*3 matrices
::NAP::51967-9298  f32  MissingValue: NaN  References: 0
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
20.0 20.5 20.6
21.0 20.0 20.0
```

14.8.5 Indexing of Matrices

```
% # Define matrix m with coordinate variables
% nap "m = {
{6.1 1 -2 9}
{2#3 0 1.2}
}"
::NAP::46-46
% $m set coo "{10 20}" "50 .. 80 ... 10"
% $m a
::NAP::46-46  f64  MissingValue: NaN  References: 1  Unit: (NULL)
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: ::NAP::48-48
Dimension 1   Size: 4      Name: (NULL)    Coordinate-variable: ::NAP::56-56
Value:
 6.1  1.0 -2.0  9.0
 3.0  3.0  0.0  1.2
% [nap "coordinate_variable(m, 0)"] a
::NAP::48-48  i32  MissingValue: -2147483648  References: 1  Unit: (NULL)
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Value:
10 20
% [nap "cv(m, 1)"] a; # can abbreviate 'coordinate_variable' to 'cv'
::NAP::56-56  i32  MissingValue: -2147483648  References: 1  Unit: (NULL)
Dimension 0   Size: 4      Name: (NULL)    Coordinate-variable: (NULL)
Value:
50 60 70 80
```

14.8.6 Cross-product indexing to extract single element (scalar)

```
% [nap m(0,0)]
6.1
% [nap m(1,2)]
```

```
0
% [nap m(1,-1)]
1.2
% [nap "m(@20, @70)"]; # Indirect indexing via coordinate variables
0
% [nap "m(@20, @75)"]; # Interpolate -- 0.5 * (0 + 1.2)
0.6
% [nap "m(0, @75)"];   # Interpolate -- 0.5 * (-2 + 9)
3.5
% [nap "m(@15, @75)"]; # Interpolate -- 0.5 * (0.6 + 3.5)
2.05
% [nap "m(@@16, @@74)"]; # Nearest
0
```

14.8.7 Cross-product indexing to extract multiple elements

```
% [nap "m(0, {2 0 -1})"]
-2 6.1 9
% [nap "m({1 0}, {2 0 -1})"]
 0.0  3.0  1.2
-2.0  6.1  9.0
% [nap "m(2, 0 .. 2)"]
6.1 1 -2
% [nap "m(1,)"];
3 3 0 1.2
% [nap "m(1, -)"]; # niladic '-' means 'reverse'
1.2 0 3 3
% [nap "m(-, -)"];
 1.2  0.0  3.0  3.0
 9.0 -2.0  1.0  6.1
```

14.8.8 Full indexing to extract single element

```
% [nap "m({0 0})"]
6.1
% [nap "m({1 2})"]
0
% [nap "m{1 -1}"]; # parentheses redundant
1.2
```

14.8.9 Full indexing to form a new array from randomly selected elements of an existing array

```
% [nap "m({{0 0}{1 1}})"]; # diagonal elements
6.1 3
% [nap "m{{0 0}{1 2}{1 -1}}"]
6.1 0 1.2
% [nap "m({
{{1 0}{2 1}{0 0}}
{{0 0}{1 3}{0 3}}
})"]
3.0 1.0 6.1
6.1 1.2 9.0
```

14.9 Linear algebra

14.9.1 Solving System of Linear Equations

Solve the following system of 2 linear equations in 2 unknowns:

$$\begin{aligned} x - 2y &= -5 \\ 2x - y &= -1 \end{aligned}$$

```
% nap "A = {
{1 -2}
{2 -1}
}"
::NAP::51972-9971
% $A
 1 -2
 2 -1
% nap "B = {-5 -1}"
::NAP::51975-9977
% $B
-5 -1
% nap "x = solve_linear(A, B)"
::NAP::51979-9257
% $x a
::NAP::51979-9257  f64  MissingValue: NaN  References: 1  Unit: (NULL)
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Value:
1 3
```

14.9.2 Inner-product operator (‘.’)

The inner-product operator (’ .’) gives the *matrix* product if the operands are matrices. Thus we can use it as follows to check the above result:

```
% [nap "A . x"]
-5 -1
```

It gives the *dot* (*scalar*) product if the operands are vectors. Here are some examples:

```
% [nap "w = {1r4 1r2 1r4}"]
0.25 0.5 0.25
% [nap "x = {3 2 4}"]
3 2 4
% [nap "w . x"]
2.75
% [nap "sum(w * x)"]; # same as this
2.75
% # i.e. weighted-sum or 'vector dot-product'
```

14.9.3 Linear Regression

The following defines the least-squares straight line using function `regression`:

```
% nap "x = {1 3 4 6 8 9 11 14}"
::NAP::14-14
% nap "y = {1 2 4 4 5 7  8  9}"
::NAP::16-16
% nap "c = regression(x, y)"
```

```
::NAP::31-31
% $c
0.545455 0.636364
% nap "y_intercept = c(0)"
::NAP::34-34
% nap "slope = c(1)"
::NAP::37-37
% nap "yy = y /// y_intercept + slope * x"; # actual = row 0, predicted = row 1
::NAP::42-42
% $yy set coo "" x
% $yy a
::NAP::42-42  f64  MissingValue: NaN  References: 1  Unit: (NULL)
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 8      Name: x         Coordinate-variable: ::NAP::14-14
Value:
1.00000 2.00000 4.00000 4.00000 5.00000 7.00000 ..
1.18182 2.45455 3.09091 4.36364 5.63636 6.27273 ..
% plot_nao yy -dash {{ } {}} -symbols {circle {}}
.win0
```

14.9.4 Multiple Regression

The following uses function `regression` to define the least-squares regression equation that predicts y from x_1 and x_2.

```
% nap "x1 = {3 5 6 8 12 14}"
::NAP::45-45
% nap "x2 = {16 10 7 4 3 2}"
::NAP::47-47
% nap "y = {90 72 54 42 30 12}"
::NAP::49-49
% nap "c = regression(transpose(x1 /// x2), y)"; # Define coefficients of regression equation
::NAP::69-69
% $c
61.4 -3.64615 2.53846
% [nap "{1 10 6} . c"]; # Use to predict y for x1 = 10, x2 = 6
40.1692
```

14.9.5 Least-squares Polynomial

The following uses function `fit_polynomial` to define the least-squares parabola that predicts y from x. It then uses function `polynomial` to predict y at two values of x.

```
% nap "x = 20 .. 70 ... 10"
::NAP::20-20
% nap "y = {54 90 138 206 292 396}"
::NAP::22-22
% nap "c = fit_polynomial(x, y, 2)"; # Define coefficients of parabola (polynomial of order 2)
::NAP::59-59
% $c
41.7714 -1.09571 0.0878571
% [nap "polynomial(c, {45 80})"]; # Use to predict y for x = 45, x = 80
170.375 516.4
```

14.10 Input/Output

14.10.1 ASCII input/output

```
% # Create some data to write out
% nap "mat = {
{1.3 -2.9 6.8}
{0.9  5.2 8.1}
}"
::NAP::13-13
% nap "latitude = {-40 -30}"
::NAP::15-15
% nap "longitude = 130 .. 150 ... 10"
::NAP::23-23
% $latitude set unit degrees_north
% $longitude set unit degrees_east
% $mat set unit degC
% $mat set coo latitude longitude
% plot_nao mat -type z
.win1
%
% # Write to ASCII file 'demo.txt'
% set f [open demo.txt w]
file4
% puts $f [$mat value]
% close $f
% gets_file demo.txt
 1.3 -2.9  6.8
 0.9  5.2  8.1

% # Read matrix from ASCII file
% nap "in = gets_matrix('demo.txt')"
::NAP::4622-4622
% $in a
::NAP::4622-4622  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:
 1.3 -2.9  6.8
 0.9  5.2  8.1
%
% # Reading data which does not form matrix
% nap "in = {[gets_file demo.txt]}"
::NAP::4629-4629
% $in a
::NAP::4629-4629  f64  MissingValue: NaN  References: 1  Unit: (NULL)
Dimension 0   Size: 6      Name: (NULL)    Coordinate-variable: (NULL)
Value:
1.3 -2.9 6.8 0.9 5.2 8.1
```

14.10.2 netCDF input/output

```
% # Create some data to write out (same as in ascii.txt)
% nap "mat = {
{1.3 -2.9 6.8}
{0.9  5.2 8.1}
```

```
}"
::NAP::13-13
% nap "latitude = {-40 -30}"
::NAP::15-15
% nap "longitude = 130 .. 150 ... 10"
::NAP::23-23
% $latitude set unit degrees_north
% $longitude set unit degrees_east
% $mat set unit degC
% $mat set coo latitude longitude
% plot_nao mat
.win0
% plot_nao mat -type z
.win1
%
; # netCDF output to new file written with single command
% $mat netcdf demo.nc temperature
::NAP::4641-4641
;
; # Following will fail if you don't have ncdump installed.
; # You can install it under windows by unzipping file
; # /sol/home/dav480/tcl_nap/install/windows/netcdf-3.5.0.win32bin.ZIP
; # into say C:\Program Files\netcdf
; # You need to include the bin (e.g. C:\Program Files\netcdf\bin) in
; # environment variable PATH (which you set using Control Panel: system: advanced).
;
% ncdump demo.nc
netcdf demo {
dimensions:
        latitude = 2 ;
        longitude = 3 ;
variables:
        int latitude(latitude) ;
                latitude:units = "degrees_north" ;
        int longitude(longitude) ;
                longitude:units = "degrees_east" ;
        double temperature(latitude, longitude) ;
                temperature:_FillValue = nan ;
                temperature:units = "degC" ;
data:

 latitude = -40, -30 ;

 longitude = 130, 140, 150 ;

 temperature =
  1.3, -2.9, 6.8,
  0.9, 5.2, 8.1 ;
}
;
# netCDF input of whole variable
% nap "in = [nap_get netcdf demo.nc temperature]"
::NAP::14406-4855
% $in a
::NAP::14406-4855  f64  MissingValue: NaN  References: 1  Unit: degC
Dimension 0   Size: 2      Name: latitude  Coordinate-variable: ::NAP::14408-5607
```

```
Dimension 1   Size: 3      Name: longitude  Coordinate-variable: ::NAP::14409-5114
Value:
 1.3 -2.9  6.8
 0.9  5.2  8.1
% [nap cv(in,0)] a
::NAP::14408-5607  i32  MissingValue: -2147483648  References: 1  Unit: degrees_north
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Value:
-40 -30
% [nap cv(in,1)] a
::NAP::14409-5114  i32  MissingValue: -2147483648  References: 1  Unit: degrees_east
Dimension 0   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
130 140 150
;
% # Change row 0 in variable 'temperature' of file 'demo.nc'
% [nap "{-8 0 -1}"] netcdf -index "0, 0 .. 2" demo.nc temperature
% exec ncdump demo.nc
netcdf demo {
dimensions:
        latitude = 2 ;
        longitude = 3 ;
variables:
        int latitude(latitude) ;
                latitude:units = "degrees_north" ;
        int longitude(longitude) ;
                longitude:units = "degrees_east" ;
        double temperature(latitude, longitude) ;
                temperature:_FillValue = nan ;
                temperature:units = "degC" ;
data:

 latitude = -40, -30 ;

 longitude = 130, 140, 150 ;

 temperature =
  -8, 0, -1,
  0.9, 5.2, 8.1 ;
}
%
% # Read columns 0 and 2 from variable 'temperature' of file 'demo.nc'
% nap "in = [nap_get netcdf demo.nc temperature ",{0 2}"]"
::NAP::14436-4716
% $in a
::NAP::14436-4716  f64  MissingValue: NaN  References: 1  Unit: degC
Dimension 0   Size: 2      Name: latitude  Coordinate-variable: ::NAP::14441-4715
Dimension 1   Size: 2      Name: longitude  Coordinate-variable: ::NAP::14444-6978
Value:
-8.0 -1.0
 0.9  8.1
```

14.10.3 CIF Input/output

```
% # Create some data to write out (same as in ascii.txt)
% nap "mat = {
```

```
{1.3 -2.9 6.8}
{0.9  5.2 8.1}
}"
::NAP::13-13
% nap "latitude = {-40 -30}"
::NAP::15-15
% nap "longitude = 130 .. 150 ... 10"
::NAP::23-23
% $latitude set unit degrees_north
% $longitude set unit degrees_east
% $mat set unit degC
% $mat set coo latitude longitude
% plot_nao mat
.win0
% plot_nao mat -type z
.win1
%
; # Write to CIF
%
% $mat a
::NAP::13-13  f64  MissingValue: NaN  References: 1  Unit: degC
Dimension 0   Size: 2      Name: latitude  Coordinate-variable: ::NAP::15-15
Dimension 1   Size: 3      Name: longitude  Coordinate-variable: ::NAP::1925-1925
Value:
 1.3 -2.9  6.8
 0.9  5.2  8.1
% put_cif mat demo.cif; # Write CIF
% # You can use GUI CIF browser to check this new file 'demo.cif'
%
% # Read from CIF
% nap "in = [get_cif demo.cif]"
::NAP::17738-5075
% $in a
::NAP::17738-5075  f32  MissingValue: NaN  References: 1  Unit: (NULL)
Dimension 0   Size: 2      Name: latitude  Coordinate-variable: ::NAP::17597-9999
Dimension 1   Size: 3      Name: longitude  Coordinate-variable: ::NAP::17669-9608
Value:
 1.3 -2.9  6.8
 0.9  5.2  8.1
```

14.11 Defining Nap functions

The final command in the procedure should be a Nap command defining the result.

The following procedure defines a Nap function `sind(`x`)` to be sin(*angleInDegrees*):

```
% proc sind x {nap "sin(x / 180p-1)"}
% [nap "sind {0 30 90}"]; # Try it
0 0.5 1
```

The following procedure defines a Nap function giving zonal means from a matrix with latitude/longitude coordinate variables:

```
% proc zonal_mean matrix {
      nap "latitude  = coordinate_variable(matrix, 0)"
      nap "longitude = coordinate_variable(matrix, 1)"
      nap "a = area_on_globe(latitude, longitude)"
```

```
      nap "sum(a * matrix, 1) / sum (a, 1)"
}
% pwd
/sol/home/dav480/tcl_nap/data
% nap "z = [nap_get net zht_r21.nc zht]"
::NAP::35-35
% nap "zm = zonal_mean(z)"
::NAP::201-201
% $zm a
::NAP::201-201  f64  MissingValue: NaN  References: 1  Unit: (NULL)
Dimension 0   Size: 56     Name: latitude  Coordinate-variable: ::NAP::47-47
Value:
2320.95 2270.02 2165.5 1940.87 1534.45 1004.65 ..
% plot_nao zm
.win0
```

14.12 Statistics

14.12.1 Elementary Descriptive Statistics (Toy example)

```
% nap "score = f32 {56 75 47 99 49}"
::NAP::38995-932
% $score all
::NAP::38995-932  f32  MissingValue: NaN  References: 1  Unit: (NULL)
Dimension 0   Size: 5      Name: (NULL)    Coordinate-variable: (NULL)
Value:
56 75 47 99 49
% [nap "am(score)"]; # arithmetic mean
65.2
% [nap "range(score)"]; # min, max
47 99
% [nap "sd(score)"]; # standard deviation
19.5796
% [nap "median(score)"]
56
% [nap "gm(score)"]; # geometric mean
62.5512
% [nap "rms(score)"]; # root mean square
68.0764
% [nap "percentile(score, 0 .. 100 ... 25)"]; # quartiles
47 49.082 56.0391 75.1836 99
% [nap "# {4 1 3 4 0 0 4}"]; # frequencies of 0, 1, 2, ...
2 1 0 1 3
```

14.12.2 Example using real satellite data

```
cd /sol/home/dav480/tcl_nap/data
% nap "c1 = [nap_get netcdf modis.nc c1]"; # Read channel 1.  (Could use GUI)
::NAP::42178-9292
% $c1 head
::NAP::42178-9292  f32  MissingValue: NaN  References: 1  Unit: (NULL)
Dimension 0   Size: 532    Name: latitude  Coordinate-variable: ::NAP::42179-78
Dimension 1   Size: 742    Name: longitude  Coordinate-variable: ::NAP::42180-2473
% nap "zm_c1 = am(c1,1)"; # zonal mean of channel 1
::NAP::42200-98
% $zm_c1 a; # a = all
```

```
::NAP::42200-98  f64  MissingValue: NaN  References: 1  Unit: (NULL)
Dimension 0   Size: 532    Name: latitude  Coordinate-variable: ::NAP::42179-78
Value:
_ 2367.32 2516.54 2522.22 2543.68 2528.35 ..
% plot_nao zm_c1
.win10
% nap "c2 = [nap_get netcdf modis.nc c2]"; # Read channel 2.
::NAP::43687-4396
% plot_nao "am(c2,1)" -title "zonal mean of channel 2"
.win11
% [nap "moving_correlation(c1, c2)"]; # r = 0.908556 based on n = 287695
     0.908556

287695.000000
% nap "dif = c2 - c1"; # difference between channels
::NAP::45219-9984
% plot_nao dif
.win12
% plot_nao "am(dif,1)" -title "zonal mean of c2-c1"
.win13
```