The NAP (N-Dimensional Array Processor) Extension to Tcl

Harvey Davies
CSIRO Atmospheric Research
Private Bag Number 1, Aspendale
Victoria 3195, Australia
harvey.davies@csiro.au

ABSTRACT

NAP is a loadable extension of Tcl that provides a convenient, powerful and efficient facility for processing data in the form of n-dimensional arrays. Special facilities are 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.

1. INTRODUCTION

After developing a query language *FAN* [1] for array-oriented files, the author became convinced that a proper scripting language was needed for the flexible mathematical processing of large volumes of array-oriented data such as the results of atmospheric modeling typified by those of Hunt and Davies[8].

The author then became involved in the development of the *CAPS* [17][16] software for processing satellite data. This software is based on Tcl/Tk and consists of two new extensions written in C, *CAPS* and *NAP*. CAPS is based on NAP but NAP can be used independently of CAPS and has been released as free software from the sourceForge facility. (See [3].)

NAP provides the essence of array-processing languages such as APL, J [9], IDL [14], DPML [5] and Matlab [15]. NAP has a number of innovative features including support for grid-oriented data based on continuous spatial coordinates.

NAP provides these facilities with a distinct *Tcl flavor*. NAP sets standard Tcl variables to the (string value of) identifying handles of the internal structures used to store the arrays. Each such structure has a Tcl command associated with it. This command is used to modify the structure and obtain information from it.

Careful coding of NAP has achieved speeds close to those of pure C code, despite the need to test for missing values of all operands. Efficient use of memory is achieved by providing a range of data-types and automatically deleting data which is no longer needed.

2. TYPOGRAPHIC CONVENTIONS

The examples in this paper have input command lines beginning with the standard Tcl prompt '`%`'. Some output has been edited to reduce its width to that available.

Syntax is specified using the following conventions:
Anything enclosed in brackets ('[' and ']') is optional.
Alternatives are separated by a vertical bar ('|').

3. FEATURES OF NAP

NAP data are stored in C structures called *n-dimensional array objects* or *NAOs*. These include information such as data-type, *OOC-name* (identifying handle generated by NAP), label, unit of measure, reference count, missing-value (value indicating null data), rank (number of dimensions), dimension sizes, dimension names and 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.

A *coordinate-variable* is a vector defining the relationship between subscript values and distances along an underlying continuous physical dimension. For example a geographic matrix might have two coordinate variables defining the latitude of each row and the longitude of each column.

The main NAP command `nap` evaluates an expression based on C conventions (like the Tcl `expr` command). Parsing is done using the *GNU bison* implementation of *yacc*[10].

NAP arithmetic has the following significant features.

Operand shapes are compatible provided the trailing dimensions match. (E.g. one can add a 3-element vector to a 4×3 matrix.) Operands with inappropriate data types are automatically converted to the most space-efficient types possible without danger of data loss. (E.g. attempting to add a 32-integer to a 32-float would result in both being converted to 64-bit floats.)

Infinity arithmetic [2] (using ∞ and NaN) is supported and is based on the IEEE standard 754 for binary floating-point arithmetic described by Goldberg[6]. If an operand's value is NaN or matches its missing-value then the result is set to the missing-value.

Each NAO can be accessed via its own Tcl command called an *object-oriented command* or *OOC*. The first argument of an OOC specifies a *method*. If there are no arguments (default method) then data are displayed. There are methods to

- display data and other information from the NAO
- write data to files and BLT vectors
- modify data and other properties

It is seldom necessary to explicitly specify OOC-names. Within expressions, they can be replaced by variable names. OOCs can be executed using the '`$`' and '`[]`' of Tcl syntax.

Unlike `expr`, `nap` has an assignment operator ('='). This sets a Tcl variable to an OOC-name, increments the reference count and puts a trace on this variable to decrement the count if this variable is deleted or changed and then delete the NAO if the count drops to zero. Reference counting is also automatic for the various kinds of pointer from one NAO to another. However it is occasionally necessary to explicitly adjust reference counts for other types of reference such as those from Tk widgets.

NAP includes interfaces to the HDF[12] and netCDF[18] file formats. Both input and output are supported.

There is a new photo image format for NAP data. This can be used to display data as images and write it to image files with standard formats (e.g. GIF, JPEG).

The user can define functions using code written in Tcl, C or Fortran.

4. COMPARISON WITH ALTERNATIVES

Standard Tcl has an array facility based on associative arrays. This is not a *data parallel* facility and it is therefore necessary to process each element individually by looping. This is tedious to code and far too slow for serious mathematical processing of arrays containing millions of elements.

Three extensions provide array processing facilities which partially overcome these problems. The following sections compare these with standard Tcl and NAP. This comparison is based on the following very simple standard problem:

1. Assign to x a vector containing 2, 2.5 and 5.
2. Assign to y the vector containing the squares of these.
3. Display y.

4.1 Standard Tcl Array Facility

This standard problem can be done as follows:

```
% set x(0) 2
2
% set x(1) 2.5
2.5
% set x(2) 5
5
% set n [array size x]
3
% for {set i 0} {$i < $n} {incr i} {
    set y($i) [expr "$x($i) * $x($i)"]
}
% for {set i 0} {$i < $n} {incr i} {
    puts "$y($i)"
}
4
6.25
25
```

One can handle multiple dimensions by using (text) indices consisting of subscripts separated by commas, as in

```
% set matrix(2,3) 2.5
```

4.2 BLT vector Command

Howlett[7] describes the BLT vector facility which operates on internal structures containing 1-dimensional 64-bit floating-point values. The following example shows how this facility can be used to solve the above standard problem:

```
% vector create x 3; # create 3-element vector x
::3
% x set {2 2.5 5};# store these values in it
% x dup y;# create new vector y as copy of x
% y expr "x * x";# multiply x by x & store in y
% set y(:);# display y
4.0 6.25 25.0
```

4.3 tk3D tensor package 'tns'

McKay[11] describes *tns* which was developed by General Motors. It provides arrays (tensors) of six data types and any rank. The standard problem could be done as follows:

```
% tensor x -initial {2 2.5 5};# create x
% tensor y -size {3};# create 3-element y
% y = tensor x;# copy x to y
% y *= tensor x;# multiply y by x
% y;# display y
4.0 6.25 25.0
```

4.4 'TiM' image/matrix processing extension

Thiébaut[4] describes *TiM* which was developed at the Observatoire de Lyon in France. This has five data types. All data are treated as matrices, so a scalar is represented by a 1×1 matrix and a vector by a $1 \times n$ (or $n \times 1$) matrix. The astronomical *FITS* file format is supported. Our standard problem could be done as follows:

```
% @set x {[2.0,2,5,5.0]}
% @set y "$x^2"
% @print $y
4.0 6.25 25.0
```

Note that the `@set` command sets a Tcl variable to an array handle in a similar fashion to NAP.

4.5 NAP

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

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

5. SAMPLE SESSION

The following sample session continues the above example to illustrate further basic features of NAP.

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 to NAOs.

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. These string values in the above example 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 MisVal: NaN Refs: 1 Unit: (NULL)
Dimension 0 Size: 3 Name: (NULL) CoordVar: (NULL)
Value:
2 2.5 5
```

This illustrates the '`all`' *method* which provides a more detailed description of the NAO than the default method.

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 NAO's reference count was zero, as it was not referenced by anything (e.g. a Tcl variable). So the NAO and its associated OOC were automatically deleted after the first execution of the OOC.

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

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

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

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

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

The following four commands respectively illustrate:

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

```
% [nap "score(2)"] all
::NAP::20-20 f32 MisVal: NaN Refs: 0 Unit: (NULL)
Value:
47
% [nap "score({2 0 4})"] all
::NAP::25-25 f32 MisVal: NaN Refs: 0 Unit: (NULL)
Dimension 0 Size: 3 Name: (NULL) CoordVar: (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 'Σ'
2. function `nels`, which gives the *number of elements*
3. the use of these to calculate an *arithmetic-mean*

```
% [nap "sum(score)"]
326
% [nap "nels(score)"]
5
% [nap "sum(score) / nels(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)/nels(x)"}
% [nap "mean(score)"]
65.2
```

Functions `min` and `max` operate in a manner similar to `sum`, as shown by:

```
% [nap "min(score)"]
47
% [nap "max(score)"]
99
```

If the function `reshape` has two arguments then it reshapes the first to the shape specified by the second, as in:

```
% [nap "reshape({1.3 9.2 -1 0}, {2 3})"]
 1.3  9.2 -1.0
 0.0  1.3  9.2
```

Here a 4-element vector is reshaped to a 2 × 3 matrix. Note the recycling after the 4 elements were exhausted.

If there is only one argument then this is reshaped to a vector with the same number of elements. E.g.

```
[nap "reshape({{1 3 2}{0 -9 7}})"]
1 3 2 0 -9 7
```

Procedures defining NAP functions have arguments and results whose values within the procedure 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 "max(reshape(n)) > 1"]]} {
        nap "n > 1 ? n * factorial(n-1) : 1"
    } else {
        nap "1"
    }
}
% [nap "factorial {4 1 6 0}"]
24 1 720 1
```

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`'.

6. DATA TYPES

NAP provides the following data types:

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

7. NAP COMMANDS

NAP includes three ordinary Tcl commands. NAOs are created by the commands `nap`, which executes a specified expression, and `nap_get`, which reads several varieties of file. The `nap_info` command provides information about the NAP system.

7.1 Reading Files using 'nap_get' Command

The `nap_get` command reads data from a file and uses this data to create a NAO. The first argument specifies the type of file, which can be `binary`, `hdf` or `netcdf`.

7.1.1 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 that of a vector whose size matches that of the file.

The following example first writes six 32-bit floating-point values to a file using standard Tcl commands. This data is then read back into a NAO named '`in`' using '`nap_get binary`':

```
% set file [open float.dat w]
file4
% set data {1.5 -3.0 0 2 4 5}
1.5 -3 0 2 4 5
% puts -nonewline $file [binary format f* $data]
% close $file
% set file [open float.dat]
file4
% nap "in = [nap_get binary $file f32]"
::NAP::13-13
% close $file
% $in all
::NAP::13-13 f32 MisVal: NaN Refs: 1 Unit: (NULL)
Dimension 0 Size: 6 Name: (NULL) CoordVar: (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 MisVal: NaN Refs: 0 Unit: (NULL)
Dimension 0 Size: 2 Name: (NULL) CoordVar: (NULL)
Dimension 1 Size: 3 Name: (NULL) CoordVar: (NULL)
Value:
 1.5 -3.0  0.0
 2.0  4.0  5.0
% close $file
```

7.1.2 Reading netCDF and HDF Data

NetCDF and HDF files are read using the command
`nap_get netcdf|hdf` *filename name* [*index*]
name is the name of a variable or attribute and has the form

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

If *index* is omitted then the entire variable is read in. Otherwise *index* selects (using cross-product indexing if multidimensional) elements from the file.

7.2 Object-Oriented Commands

Object-Oriented Commands (OOCs) are used to:

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

The first argument specifies the *method*.

Method `value` returns data values. If there are no switches then *all* values are included, as in:

```
% [nap "2 ** (0 .. 12)"] value
1 2 4 8 16 32 64 128 256 512 1024 2048 4096
```

The default method (without switches) displays only the first 20 lines and 6 columns. Thus if `value` is deleted from this command we get:

```
% [nap "2 ** (0 .. 12)"]
1 2 4 8 16 32 ..
```

Method `all` returns both data values and metadata. E.g.

```
% [nap "2 ** (0 .. 12)"] all
::NAP::21-21 f32 MisVal: NaN Refs: 0 Unit: (NULL)
Dimension 0 Size: 13 Name: (NULL) CoordVar: (NULL)
Value:
1 2 4 8 16 32 ..
```

All these methods (which return data values) take the following switches:
`-format` *format*: C format (default: `""` meaning automatic)
`-columns` *int*: max. no. columns (`-1`: no limit)
`-lines` *int*: max. no. lines (`-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

Thus the following gives one decimal place and ten columns using the default method:

```
% [nap "2 ** (0 .. 12)"] -f %.1f -c 10
1.0 2.0 4.0 8.0 16.0 32.0 64.0 128.0 256.0 512.0 ..
```

Most NAO components have a corresponding method providing their value. The following example illustrates three of these metadata methods; `datatype`, `shape` and `missing`:

```
% [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
```

Some of these NAO components can be modified using the `set` method. The following continuation of this example shows how changing the missing value to `-9` changes the sums of the columns:

```
% [nap "sum(x)"]
3.6 4.4 -8
% $x set missing -9
% $x missing
-9
% [nap "sum(x)"]
3.6 4.4 1
```

Note that the `-9` element is no longer included in the sum.

The `set` method can also be used to modify data values. The following changes the value of two elements:

```
% $x set value "{-1 -3}" "1,{0 2}"
% $x
 0.0  2.4  1.0
-1.0  2.0 -3.0
```

Methods '`write`', '`netcdf`' and '`hdf`' write binary data from the NAO to a file. The following example writes this 2 × 3 matrix to file `y.tmp` and then reads it back (as a 6-element vector).

```
% set file [open x.tmp w]
file4
% $x write $file
% close $file
% set file [open x.tmp]
file4
% [nap_get binary $file f64]
0 2.4 1 -1 2 -3
% close $file
```

8. NAP EXPRESSIONS

8.1 Constants

NAP provides a 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.

8.1.1 Integer Scalar Constants

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

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

Examples of integer scalar constants are:

Constant	**Decimal Value**	**Explanation**	**Data Type**
`14`	14		`i32`
`14u8`	14		`u8`
`014`	12	octal	`u32`
`014i8`	12	octal	`i8`
`0x14`	20	hexadecimal	`u32`
`_`	-2147483648	-2^{31} (missing)	`i32`

8.1.2 Floating-point Scalar Constants

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

A mantissa can be written in either decimal or rational form. A rational mantissa consists of two integers separated by `r` and represents their ratio.

The letter `e` indicates an exponent with base 10. The letter `p` indicates an exponent with base π.

Examples of floating-point scalar constants are:

Constant	**Displayed Value**	**Explanation**	**Data Type**
`4.0`	4		`f64`
`4f32`	4		`f32`
`2r3`	0.666667	$\frac{2}{3}$	`f64`
`1e4`	10000	1×10^4	`f64`
`1p1`	3.14159	$1 \times \pi^1$	`f64`
`180p-1`	57.2958	$180 \times \pi^{-1}$	`f64`
`1r3p1f32`	1.0472	$\frac{1}{3} \times \pi^1$	`f32`
`1i`	Inf	∞	`f64`
`1if32`	Inf	∞	`f32`
`1n`	_	NaN	`f64`
`1nf32`	_	NaN	`f32`

8.1.3 Numeric Array Constants

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

A vector (1-dimensional array) constant is enclosed by one level of braces. An example is '`{2 -7 8}`'.

A matrix (2-dimensional array) constant is enclosed by two levels of braces. An example is '`{{1 3 5}{2 4 6}}`'.

An n-dimensional array constant is enclosed by n levels of braces.

8.1.4 String Constants

String constants are enclosed by either two apostrophes (e.g. `’Hello world’`) or two grave accents (e.g. `‘Hello world‘`). 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`.

8.2 Operators

The table on the right is essentially a superset of **Table 5.2** in Ousterhout [13]. As there, groups of operators between horizontal lines have the same precedence; higher groups have higher precedence.

Operators are left-associative unless specified otherwise.

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

‘AP’ means *arithmetic progression.*

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

The *link* operators ‘`,`’ and ‘`...`’ are identical except for precedence. ‘`,`’ is used to separate function arguments and the subscripts of cross-product indices. ‘`...`’ is used to specify the step size of arithmetic progressions, as in:

```
% [nap "3 .. 9 ... 2"]
3 5 7 9
```

The following example illustrates the difference between ‘`//`’ and ‘`///`’.

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

8.2.1 Inverse Indexing Operators ‘`@`’, ‘`@@`’ and ‘`@@@`’

These three operators take a vector left argument. The result is a subscript of this vector. The unary case will be discussed in the section on indirect indexing.

The ‘`@`’ *interpolated subscript* operator requires a sorted left argument. The result of v`@`b is the `f32` subscript s such that $v_s = b$. For example:

Syntax	**Result**
`a**b`	a^b Right-associative
`-a`	Negative of a
`!a`	Logical NOT: 1 if a is zero, else 0
`~a`	Bit-wise complement of a
`#a`	Frequencies of values 0, 1, 2, ...
`@a`	indirect subscript
`@@a`	indirect subscript
`v@b`	s such that $v_s = b$, where v is ordered vector
`v@@b`	`i32` s for which $\lvert v_s - b \rvert$ is least
`v@@@b`	smallest `i32` s for which $v_s = b$
`[a]...[b]`	Boxed vector pointing to a and b
`x..y`	AP from x to y in steps of `+1` or `-1`
`x..p`	AP from x to value pointed to by p_0 in steps of value pointed to by p_1
`u#v`	u copies of v
`p#b`	Cross-product replication
`u+*v`	(u and v vectors) Scalar (dot) product
`A+*B`	(A and B matrices) Matrix product
`a*b`	$a \times b$
`a/b`	$a \div b$
`a%b`	Remainder r 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 OR of a and b
`a&&b`	Logical AND: 1 if $a \neq 0$ and $b \neq 0$, else 0
`a||b`	Logical OR: 1 if $a \neq 0$ or $b \neq 0$, else 0
`a?b:c`	Choice: if $a \neq 0$ then b, else c
`a//b`	Concatenate along existing dimension
`a///b`	Concatenate along new dimension
`[a],[b]`	Boxed vector pointing to a and b
`n=a`	Result is a. Right-associative Side Effect: Set n to OOC-name of a

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

The result is 1.5 because 3.5 is halfway between 3.4 (subscript 1) and 3.6 (subscript 2).

The ‘@@’ *closest* operator is defined so that v@@b gives 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 ‘@@@’ *match* operator is defined so that v@@@b gives the smallest i32 subscript s for which $v_s = b$. For example:

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

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

8.2.2 The ‘#’ Operator

The unary *tally* ‘#’ operator 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.

The binary *replicate* ‘#’ operator treats the left argument as the number of required replications of the right argument. The arguments can be vectors or scalars. The result is a vector.

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

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

This works because the left-hand argument is:

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

8.3 Built-in Functions

8.3.1 Elemental Functions

An *elemental* function is one in which

- the result has the same shape as the argument(s)
- each element of the result is defined by applying the function to the corresponding element of the argument

The standard mathematical functions provided by the expr command are all provided by NAP as elemental functions. Additional elemental functions provided by NAP include:

Function	Result
isnan(x)	1 if x is NAN, else 0
random(x)	random number r such that $0 \leq r < x$
sign(x)	$(x > 0) - (x < 0)$

There are also elemental functions for data-type conversion, with the same names as the data-types. The following example uses function u8 to display the ASCII codes for ’abcdef’ and then reverses this process using function c8:

```
% [nap "u8('abcdef')"]
97 98 99 100 101 102
% [nap "c8(97 .. 102)"]
abcdef
```

8.3.2 Reduction and Scan Functions

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

The NAP reduction functions are:

Function	Result
count($x[, r]$)	Number of non-missing elements
max($x[, r]$)	Maximum
min($x[, r]$)	Minimum
prod($x[, r]$)	Product
sum($x[, r]$)	Sum

The optional second argument of reduction functions is called the *verb-rank* (as in J). It specifies the rank of the sub-arrays (cells) to which the process is applied. In the case of a matrix argument, one can specify a verb-rank of 1 to get processing of rows rather than columns.

NAP currently has only one *scan* function, psum, which produces partial sums.

8.3.3 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. An example is function shape(x) which returns the shape of x (as a vector).

8.3.4 Functions which change shape or order

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

8.3.5 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.

8.3.6 Correlation

Function correlation(x[, y, [lag_0, lag_1, $\ldots$]]) calculates Pearson product-moment correlations between either:

- two arrays of the same shape (treated as vectors)
- the columns of a matrix
- an array and a moving window within it

8.3.7 Grid Functions

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

8.3.8 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 x
prune(x)	Ragged NAO corresponding to normal x

8.4 Indexing

Indexing is the process of selecting elements of an array for extraction or modification. NAP extends this concept to the estimation (using interpolation) of values *between* the elements.

An index can appear:

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

8.4.1 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:

```
% nap "vector = {2 -5 9 4}"
::NAP::41-41
% [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 $0.5 \times 9.0 + 0.5 \times 4.0 = 6.5$ using ordinary one-dimensional linear interpolation.

8.4.2 Subscript

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

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

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

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

```
% [nap "vector(-3)"]
-5
```

8.4.3 Elemental Index

An *elemental index* of an array of rank r is a vector of r subscripts specifying an element of the 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 $0.25 \times 0.0 + 0.25 \times 7.0 + 0.25 \times (-4.0) + 0.25 \times (-9.0) = -1.5$

8.4.4 Index

An *index* is an array defining one or more elemental indices. There are three types: *shape-preserving* (index of vector), *full* and *cross-product*. *Full* and *cross-product* indices are two methods of indexing arrays with multiple dimensions.

8.4.5 Shape-Preserving Index

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 3-element vector to produce a 3-element vector
- a 2 × 3 matrix to produce a 2 × 3 matrix:

```
% $vector
2 -5 9 4
% [nap "vector 2"]
9
% [nap "vector {2 2.5 2}"]
9 6.5 9
% [nap "vector {
{1 0 2.5}
{-1 2 1}
}"]
-5.0  2.0  6.5
 4.0  9.0 -5.0
```

8.4.6 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}"]
-1.5
% [nap "mat {
    {0.5 1.5}
    {0 1}
    {-1 -1}
}"]
-1.5 0 -9
```

8.4.7 Cross-product-index

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

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

The following example selects

- rows 1 and 0
- columns 2, 0 -1 and 0

```
% $mat
 1.5  0.0  7.0
 2.0 -4.0 -9.0
% [nap "mat({1 0},{2 0 -1 0})"]
-9.0  2.0 -9.0  2.0
 7.0  1.5  7.0  1.5
```

8.4.8 Indirect Indexing

Indirect indexing simplifies the use of coordinate variable values in index expressions. For example, suppose we have temperatures at 2-hourly intervals from time 10:00 to 16:00 as follows:

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

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

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

Indirect indexing allows us to omit the left argument of operators `@` and `@@` in such expressions. Thus:

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

9. NAP LIBRARY OF TCL CODE

This library consists of about seventy Tcl procedures defining NAP functions and providing other tools for NAP. These include:

- statistics functions
- fortran binary I/O
- map projections
- `plot_nao` for visualisation of NAOs
- `make_dll` which defines a new Tcl command based on C or Fortran code

10. CONCLUSIONS

Tcl has not generally been considered suitable for large-scale mathematical processing. However the development of NAP has shown that Tcl is a suitable framework for the development of mathematical tools. Tcl variables can be used by setting them to array object handles rather than the actual data. These array objects can have associated commands to display, transmit and modify their contents.

NAP has been developed into a very flexible, efficient and user-friendly tool for the mathematical processing of large volumes of array data. NAP can be considered a language within the Tcl language, but NAP's conventions are designed to match those of Tcl and the integration into the Tcl environment works very well.

11. ACKNOWLEDGMENTS

The author would like to thank his CSIRO colleagues Peter Turner, Ian Grant and Martin Dix. Peter played an important role in the development of NAP. Ian and Martin reviewed this paper and NAP documentation. The author would also like to thank Russ Rew of the UCAR Unidata Program Center for encouraging the author's software development efforts and enabling the author to become involved in the development of netCDF.

12. REFERENCES

[1] H. L. Davies. FAN - An Array-oriented Query Language. In G. Grinstein, U. Lang, and A. Wierse, editors, *Second Workshop on Database Issues for Data Visualization*. IEEE, Oct 1995.

[2] H. L. Davies. Infinity Arithmetic, Comparisons and J. *APL Quote Quad*, 25(4):28–34, 1995.

[3] H. L. Davies. Tcl-nap Project. URL: `http://tcl-nap.sourceforge.net/`, 2002.

[4] Eric Thiébaut. TiM, a Tcl extension for image/matrix processing. URL: `http://www-obs.univ-lyon1.fr/~thiebaut/TiM/TiM.html`, 1996.

[5] R. S. Francis, I. D. Mathieson, P. G. Whiting, M. R. Dix, H. L. Davies, and L. D. Rotstayn. A Data Parallel Scientific Modelling Language. *J. of Parallel and Distributed Computing*, 21:46–60, 1994.

[6] D. Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[7] G. A. Howlett. Data Objects. In *Proceedings of the Sixth Annual Tcl/Tk Workshop*. USENIX, Sep 1998.

[8] B. G. Hunt and H. L. Davies. Mechanism of multi-decadal climatic variability in a global climate model. *International Journal of Climatology*, 17(6):565–580, 1997.

[9] K. E. Iverson. *J Introduction and Dictionary*. Iverson Software Inc., Toronto, 1996.

[10] J. R. Levine, T. Mason, and D. Brown. *lex & yacc*. O'Reilly and Asssociates, Sebastopol Calif., 2nd edition, 1992.

[11] N. D. McKay. Tcl/Tk Extensions for Visualization of Large Data Sets. In *O'Reilly Open Source Convention*. O'Reilly and Associates, Inc, July 2001.

[12] National Center for Supercomputing Applications. HDF Home Page. URL: `http://hdf.ncsa.uiuc.edu/hdf4.html`, 2002.

[13] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, USA, 1994.

[14] Research Systems, Inc. IDL. URL: `http://www.rsinc.com/idl/index.asp`, 2002.

[15] The MathWorks, Inc. The MathWorks. URL: `http://www.mathworks.com/`, 2002.

[16] P. J. Turner and H. L. Davies. Advances in CAPS. In T. R. McVicar, editor, *Proceedings of the Land EnvSat Workshop: 10th Australasian Remote Sensing Photogrammetry Conference, Adelaide*, pages 71–80, Canberra, A.C.T., Australia, 2000. CSIRO Land and Water.

[17] P. J. Turner, H. L. Davies, P. C. Tildesley, , and C. Rathbone. Common AVHRR processing software (CAPS). In T. R. McVicar, editor, *Proceedings of the Land AVHRR Workshop: 9th Australasian Remote Sensing Photogrammetry Conference, Sydney*, pages 51–58, Canberra, A.C.T., Australia, 1998. CSIRO Land and Water.

[18] Unidata Program Center, University Corporation for Atmospheric Research. NetCDF. URL: `http://www.unidata.ucar.edu/packages/netcdf/index.html`, 2002.