N-Dimensional Lists (ndlist)

Version 0.8

Alex Baker

https://github.com/ambaker1/ndlist

May 16, 2024

Abstract

The "ndlist" package is a pure-Tcl package for tensor manipulation and processing.

This package is also a Tin package, and can be loaded in as shown below:

Example 1: Installing and loading "ndlist"

Code:

```
package require tin
tin add -auto ndlist https://github.com/ambaker1/ndlist install.tcl
tin import ndlist
```

1-Dimensional Lists (Vectors)

Lists are foundational to Tcl, so in addition to providing utilities for ND-lists, this package also provides utilities for working with 1D-lists, or vectors.

Range Generator

The command *range* simply generates a list of integer values. This can be used in conjunction with the Tcl *foreach* loop to simplify writing "for" loops. There are two ways of calling this command, as shown below.

```
range $n
range $start $stop <$step>
```

$n	Number of indices, starting at 0 (e.g. 3 returns 0 1 2).
$start	Starting value.
$stop	Stop value.
$step	Step size. Default 1 or -1, depending on direction of start to stop.

Example 2: Integer range generation

Code:

```
puts [range 3]
puts [range 0 2]
puts [range 10 3 -2]
```

Output:

```
0 1 2
0 1 2
10 8 6 4
```

Example 3: Simpler for-loop

Code:

```
foreach i [range 3] {
    puts $i
}
```

Output:

```
0
1
2
```

Logical Indexing

The command *find* returns the indices of non-zero elements of a boolean list, or indices of elements that satisfy a given criterion. Can be used in conjunction with *nget* to perform logical indexing.

```
find $list <$op $scalar>
```

`$list`	List of values to compare.
`$op`	Comparison operator. Default "!=".
`$scalar`	Comparison value. Default 0.

Example 4: Filtering a list

Code:

```
set x {0.5 2.3 4.0 2.5 1.6 2.0 1.4 5.6}
puts [nget $x [find $x > 2]]
```

Output:

```
2.3 4.0 2.5 5.6
```

Linear Interpolation

The command *linterp* performs linear 1D interpolation. Converts input to double.

```
linterp $x $xList $yList
```

`$x`	Value to query in `$xList`
`$xList`	List of x points, strictly increasing
`$yList`	List of y points, same length as `$xList`

Example 5: Linear interpolation

Code:

```
puts [linterp 2 {1 2 3} {4 5 6}]
puts [linterp 8.2 {0 10 20} {2 -4 5}]
```

Output:

```
5.0
-2.92
```

Vector Generation

The command *linspace* can be used to generate a vector of specified length and equal spacing between two specified values. Converts input to double.

```
linspace $n $start $stop
```

$n	Number of points
$start	Starting value
$stop	End value

Example 6: Linearly spaced vector generation

Code:

```
puts [linspace 5 0 1]
```

Output:

```
0.0 0.25 0.5 0.75 1.0
```

The command *linsteps* generates intermediate values given an increment size and a sequence of targets. Converts input to double.

```
linsteps $step $x1 $x2 ...
```

$step	Maximum step size
$x1 $x2 ...	Targets to hit.

Example 7: Intermediate value vector generation

Code:

```
puts [linsteps 0.25 0 1 0]
```

Output:

```
0.0 0.25 0.5 0.75 1.0 0.75 0.5 0.25 0.0
```

Functional Mapping

The command *lapply* simply applies a command over each element of a list, and returns the result. The command *lapply2* maps element-wise over two equal length lists.

```
lapply $command $list $arg ...
```

```
lapply2 $command $list1 $list2 $arg ...
```

$list	List to map over.
$list1 $list2	Lists to map over, element-wise.
$command	Command prefix to map with.
$arg ...	Additional arguments to append to command after list elements.

Example 8: Applying a math function to a list

Code:

```
# Add Tcl math functions to the current namespace path
namespace path [concat [namespace path] ::tcl::mathfunc]
puts [lapply abs {-5 1 2 -2}]
```

Output:

```
5 1 2 2
```

Example 9: Mapping over two lists

Code:

```
lapply puts [lapply2 {format "%s %s"} {hello goodbye} {world moon}]
```

Output:

```
hello world
goodbye moon
```

List Statistics

The commands *max*, *min*, *sum*, *product*, *mean*, *median*, *stdev* and *pstdev* compute the maximum, minimum, sum, product, mean, median, sample and population standard deviation of values in a list. For more advanced statistics, check out the Tcllib math::statistics package.

```
max $list
```

```
min $list
```

```
sum $list
```

```
product $list
```

```
mean $list
```

```
median $list
```

```
stdev $list
```

```
pstdev $list
```

$list	List to compute statistic of.

Example 10: List Statistics

Code:

```
set list {-5 3 4 0}
foreach stat {max min sum product mean median stdev pstdev} {
    puts [list $stat [$stat $list]]
}
```

Output:

```
max 4
min -5
sum 2
product 0
mean 0.5
median 1.5
stdev 4.041451884327381
pstdev 3.5
```

Vector Algebra

The dot product of two equal length vectors can be computed with *dot*. The cross product of two vectors of length 3 can be computed with *cross*.

```
dot $a $b
```

```
cross $a $b
```

`$a`	First vector.
`$b`	Second vector.

The norm, or magnitude, of a vector can be computed with *norm*.

```
norm $a <$p>
```

`$a`	Vector to compute norm of.
`$p`	Norm type. 1 is sum of absolute values, 2 is euclidean distance, and Inf is absolute maximum value. Default 2.

Example 11: Dot and cross product

Code:

```
set x {1 2 3}
set y {-2 -4 6}
puts [dot $x $y]
puts [cross $x $y]
```

Output:

```
8
24 -12 0
```

For more advanced vector algebra routines, check out the Tcllib math::linearalgebra package.

2-Dimensional Lists (Matrices)

A matrix is a two-dimensional list, or a list of row vectors. This is consistent with the format used in the Tcllib math::linearalgebra package. See the example below for how matrices are interpreted.

$$A = \begin{bmatrix} 2 & 5 & 1 & 3 \\ 4 & 1 & 7 & 9 \\ 6 & 8 & 3 & 2 \\ 7 & 8 & 1 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 9 \\ 3 \\ 0 \\ -3 \end{bmatrix}, \quad C = \begin{bmatrix} 3 & 7 & -5 & -2 \end{bmatrix}$$

Example 12: Matrices and vectors

Code:

```
# Define matrices, column vectors, and row vectors
set A {{2 5 1 3} {4 1 7 9} {6 8 3 2} {7 8 1 4}}
set B {9 3 0 -3}
set C {{3 7 -5 -2}}
# Print out matrices (join with newline to print out each row)
puts "A ="
puts [join $A \n]
puts "B ="
puts [join $B \n]
puts "C ="
puts [join $C \n]
```

Output:

```
A =
2 5 1 3
4 1 7 9
6 8 3 2
7 8 1 4
B =
9
3
0
-3
C =
3 7 -5 -2
```

Generating Matrices

The commands *zeros*, *ones*, and *eye* generate common matrices.

```
zeros $n $m
```

```
ones $n $m
```

$n	Number of rows
$m	Number of columns

The command *eye* generates an identity matrix of a specified size.

```
eye $n
```

$n	Size of identity matrix

Example 13: Generating standard matrices

Code:

```
puts [zeros 2 3]
puts [ones 3 2]
puts [eye 3]
```

Output:

```
{0 0 0} {0 0 0}
{1 1} {1 1} {1 1}
{1 0 0} {0 1 0} {0 0 1}
```

Combining Matrices

The commands *stack* and *augment* can be used to combine matrices, row or column-wise.

```
stack $mat1 $mat2 ...
```

```
augment $mat1 $mat2 ...
```

`$mat1 $mat2 ...`	Arbitrary number of matrices to stack/augment (number of columns/rows must match)

The command *block* combines a matrix of matrices into a block matrix.

```
block $matrices
```

`$matrices`	Matrix of matrices.

Example 14: Combining matrices

Code:

```
set A [stack {{1 2}} {{3 4}}]
set B [augment {1 2} {3 4}]
set C [block [list [list $A $B] [list $B $A]]]
puts $A
puts $B
puts [join $C \n]; # prints each row on a new line
```

Output:

```
{1 2} {3 4}
{1 3} {2 4}
1 2 1 3
3 4 2 4
1 3 1 2
2 4 3 4
```

Matrix Transpose

The command *transpose* simply swaps the rows and columns of a matrix.

```
transpose $A
```

$A	Matrix to transpose, nxm.

Returns an mxn matrix.

Example 15: Transposing a matrix

Code:

```
puts [transpose {{1 2} {3 4}}]
```

Output:

```
{1 3} {2 4}
```

Matrix Multiplication

The command *matmul* performs matrix multiplication for two matrices. Inner dimensions must match.

```
matmul $A $B
```

$A	Left matrix, nxq.
$B	Right matrix, qxm.

Returns an nxm matrix (or the corresponding dimensions from additional matrices)

Example 16: Multiplying a matrix

Code:

```
puts [matmul {{2 5 1 3} {4 1 7 9} {6 8 3 2} {7 8 1 4}} {9 3 0 -3}]
```

Output:

```
24 12 72 75
```

Miscellaneous Linear Algebra Routines

The command *outerprod* takes the outer product of two vectors, $\boldsymbol{a} \otimes \boldsymbol{b} = \boldsymbol{a}\boldsymbol{b}^T$.

```
outerprod $a $b
```

`$a $b`	Vectors with lengths n and m. Returns a matrix, shape nxm.

The command *kronprod* takes the Kronecker product of two matrices, as shown in Eq. (1).

```
kronprod $A $B
```

`$A $B`	Matrices, shapes nxm and pxq. Returns a matrix, shape (np)x(mq).

$$\boldsymbol{A} \otimes \boldsymbol{B} = \begin{bmatrix} a_{11}\boldsymbol{B} & \dots & a_{1n}\boldsymbol{B} \\ \vdots & \ddots & \vdots \\ a_{n1}\boldsymbol{B} & \dots & a_{nn}\boldsymbol{B} \end{bmatrix} \tag{1}$$

Example 17: Outer product and Kronecker product

Code:

```
set A [eye 3]
set B [outerprod {1 2} {3 4}]
set C [kronprod $A $B]
puts [join $C \n]; # prints out each row on a new line
```

Output:

```
3 4 0 0 0 0
6 8 0 0 0 0
0 0 3 4 0 0
0 0 6 8 0 0
0 0 0 0 3 4
0 0 0 0 6 8
```

For more advanced matrix algebra routines, check out the Tcllib math::linearalgebra package.

Iteration Tools

The commands *zip* zips two lists into a list of tuples, and *zip3* zip three lists into a list of triples. Lists must be the same length.

```
zip $a $b
```

```
zip3 $a $b $c
```

`$a $b $c`	Lists to zip together.

Example 18: Zipping and unzipping lists

Code:

```
# Zipping
set x [zip {A B C} {1 2 3}]
set y [zip3 {Do Re Mi} {A B C} {1 2 3}]
puts $x
puts $y
# Unzipping (using transpose)
puts [transpose $x]
```

Output:

```
{A 1} {B 2} {C 3}
{Do A 1} {Re B 2} {Mi C 3}
{A B C} {1 2 3}
```

The command *cartprod* computes the Cartesian product of an arbitrary number of vectors, returning a matrix where the columns correspond to the input vectors and the rows correspond to all the combinations of the vector elements.

```
cartprod $a $b ...
```

`$a $b ...`	Arbitrary number of vectors to take Cartesian product of.

Example 19: Cartesian product

Code:

```
puts [cartprod {A B C} {1 2 3}]
```

Output:

```
{A 1} {A 2} {A 3} {B 1} {B 2} {B 3} {C 1} {C 2} {C 3}
```

N-Dimensional Lists (Tensors)

A ND-list is defined as a list of equal length (N-1)D-lists, which are defined as equal length (N-2)D-lists, and so on until (N-N)D-lists, which are scalars of arbitrary size. This definition is flexible, and allows for different interpretations of the same data. For example, the list "1 2 3" can be interpreted as a scalar with value "1 2 3", a vector with values "1", "2", and "3", or a matrix with row vectors "1", "2", and "3".

The command *ndlist* validates that the input is a valid ND-list. If the input value is "ragged", as in it has inconsistent dimensions, it will throw an error. In general, if a value is a valid for N dimensions, it will also be valid for dimensions 0 to N-1. All other ND-list commands assume a valid ND-list.

```
ndlist $nd $value
```

$nd	Rank of ND-list (e.g. 2D, 2d, or 2 for a matrix).
$value	List to interpret as an ndlist

Shape and Size

The commands *nshape* and *nsize* return the shape and size of an ND-list, respectively. The shape is a list of the dimensions, and the size is the product of the shape.

```
nshape $nd $ndlist <$axis>
```

```
nsize $nd $ndlist
```

$nd	Rank of ND-list (e.g. 2D, 2d, or 2 for a matrix).
$ndlist	ND-list to get shape/size of.
$axis	Axis to get dimension along. Blank for all.

Example 20: Getting shape and size of an ND-list

Code:

```
narray new 2D x {{1 2 3} {4 5 6}}
puts [nshape 2D [$x]]
puts [$x size]
```

Output:

```
2 3
6
```

Initialization

The command *nfull* initializes a valid ND-list of any size filled with a single value.

```
nfull $value $n ...
```

`$value`	Value to repeat
`$n ...`	Shape (list of dimensions) of ND-list.

Example 21: Generate ND-list filled with one value

Code:

```
puts [nfull foo 3 2]; # 3x2 matrix filled with "foo"
puts [nfull 0 2 2 2]; # 2x2x2 tensor filled with zeros
```

Output:

```
{foo foo} {foo foo} {foo foo}
{{0 0} {0 0}} {{0 0} {0 0}}
```

The command *nrand* initializes a valid ND-list of any size filled with random values between 0 and 1.

```
nrand $n ...
```

`$n ...`	Shape (list of dimensions) of ND-list.

Example 22: Generate random matrix

Code:

```
expr {srand(0)}; # resets the random number seed (for the example)
puts [nrand 1 2]; # 1x2 matrix filled with random numbers
```

Output:

```
{0.013469574513598146 0.3831388500440581}
```

Repeating and Expanding

The command *nrepeat* repeats portions of an ND-list a specified number of times.

```
nrepeat $ndlist $n ...
```

$value	Value to repeat
$n ...	Repetitions at each level.

Example 23: Repeat elements of a matrix

Code:

```
puts [nrepeat {{1 2} {3 4}} 1 2]
```

Output:

```
{1 2 1 2} {3 4 3 4}
```

The command *nexpand* repeats portions of an ND-list to expand to new dimensions. New dimensions must be divisible by old dimensions. For example, 1x1, 2x1, 4x1, 1x3, 2x3 and 4x3 are compatible with 4x3.

```
nexpand $ndlist $n ...
```

$ndlist	ND-list to expand.
$n ...	New shape of ND-list. If -1 is used, it keeps that axis the same.

Example 24: Expand an ND-list to new dimensions

Code:

```
puts [nexpand {1 2 3} -1 2]
puts [nexpand {{1 2}} 2 4]
```

Output:

```
{1 1} {2 2} {3 3}
{1 2 1 2} {1 2 1 2}
```

Padding and Extending

The command *npad* pads an ND-list along its axes by a specified number of elements.

```
npad $ndlist $value $n ...
```

$ndlist	ND-list to pad.
$value	Value to pad with.
$n ...	Number of elements to pad.

Example 25: Padding an ND-list with zeros

Code:

```
set a {{1 2 3} {4 5 6} {7 8 9}}
puts [npad $a 0 2 1]
```

Output:

```
{1 2 3 0} {4 5 6 0} {7 8 9 0} {0 0 0 0} {0 0 0 0}
```

The command *nextend* extends an ND-list to a new shape by padding.

```
nextend $ndlist $value $n ...
```

$ndlist	ND-list to extend.
$value	Value to pad with.
$n ...	New shape of ND-list.

Example 26: Extending an ND-list to a new shape with a filler value

Code:

```
set a {hello hi hey howdy}
puts [nextend $a world -1 2]
```

Output:

```
{hello world} {hi world} {hey world} {howdy world}
```

Flattening and Reshaping

The command *nflatten* flattens an ND-list to a vector.

```
nflatten $nd $ndlist
```

$nd	Rank of ND-list (e.g. 2D, 2d, or 2 for a matrix).
$ndlist	ND-list to flatten.

Example 27: Reshape a matrix to a 3D tensor

Code:

```
set x [nflatten 2D {{1 2 3 4} {5 6 7 8}}]
puts [nreshape $x 2 2 2]
```

Output:

```
{{1 2} {3 4}} {{5 6} {7 8}}
```

The command *nreshape* reshapes a vector into specified dimensions. Sizes must be compatible.

```
nreshape $vector $n ...
```

$vector	Vector (1D-list) to reshape.
$n ...	Shape (list of dimensions) of ND-list.

Example 28: Reshape a vector to a matrix

Code:

```
puts [nreshape {1 2 3 4 5 6} 2 3]
```

Output:

```
{1 2 3} {4 5 6}
```

Index Notation

This package provides generalized N-dimensional list access/modification commands, using an index notation parsed by the command *::ndlist::ParseIndex*, which returns the index type and an index list for the type.

::ndlist::ParseIndex $n $input

$n	Number of elements in list.
$input	Index input. Options are shown below:
:	All indices
$start:$stop	Range of indices (e.g. 0:4 or 1:end-2).
$start:$step:$stop	Stepped range of indices (e.g. 0:2:-2 or 2:3:end).
$iList	List of indices (e.g. {0 end-1 5} or 3).
$i*	Single index with a asterisk, "flattens" the ndlist (e.g. 0* or end-3*).

Additionally, indices get passed through the *::ndlist::Index2Integer* command, which converts the inputs "end", "end-integer", "integer±integer" and negative wrap-around indexing (where -1 is equivalent to "end") into normal integer indices. Note that this command will return an error if the index is out of range.

::ndlist::Index2Integer $n $index

$n	Number of elements in list.
$index	Single index.

Example 29: Index Notation

Code:

```
set n 10
puts [::ndlist::ParseIndex $n :]
puts [::ndlist::ParseIndex $n 1:8]
puts [::ndlist::ParseIndex $n 0:2:6]
puts [::ndlist::ParseIndex $n {0 5 end-1}]
puts [::ndlist::ParseIndex $n end*]
```

Output:

```
A {}
R {1 8}
L {0 2 4 6}
L {0 5 8}
S 9
```

Access

Portions of an ND-list can be accessed with the command *nget*, using the index parser *::ndlist::ParseIndex* for each dimension being indexed. Note that unlike the Tcl *lindex* and *lrange* commands, *nget* will return an error if the indices are out of range.

```
nget $ndlist $i ...
```

$ndlist	ND-list value.
$i ...	Index inputs, parsed with *::ndlist::ParseIndex*.

Example 30: ND-list access

Code:

```
set A {{1 2 3} {4 5 6} {7 8 9}}
puts [nget $A 0 :]; # get row matrix
puts [nget $A 0* :]; # flatten row matrix to a vector
puts [nget $A 0:1 0:1]; # get matrix subset
puts [nget $A end:0 end:0]; # can have reverse ranges
puts [nget $A {0 0 0} 1*]; # can repeat indices
```

Output:

```
{1 2 3}
1 2 3
{1 2} {4 5}
{9 8 7} {6 5 4} {3 2 1}
2 2 2
```

Modification

A ND-list can be modified by reference with *nset*, and by value with *nreplace*, using the index parser *::ndlist::ParseIndex* for each dimension being indexed. Note that unlike the Tcl *lset* and *lreplace* commands, the commands *nset* and *nreplace* will return an error if the indices are out of range. If all the index inputs are "`:`" except for one, and the replacement list is blank, it will delete values along that axis by calling *nremove*. Otherwise, the replacement ND-list must be expandable to the target index dimensions.

```
nset $varName $i ...  $sublist
```

```
nreplace $ndlist $i ...  $sublist
```

`$varName`	Variable that contains an ND-list.
`$ndlist`	ND-list to modify.
`$i ...`	Index inputs, parsed with *::ndlist::ParseIndex*.
`$sublist`	Replacement list, or blank to delete values.

Example 31: Replace range with a single value

Code:

```
puts [nreplace [range 10] 0:2:end 0]
```

Output:

```
0 1 0 3 0 5 0 7 0 9
```

Example 32: Swapping matrix rows

Code:

```
set a {{1 2 3} {4 5 6} {7 8 9}}
nset a {1 0} : [nget $a {0 1} :]; # Swap rows and columns (modify by reference)
puts $a
```

Output:

```
{4 5 6} {1 2 3} {7 8 9}
```

Removal

The command *nremove* removes portions of an ND-list at a specified axis.

```
nremove $ndlist $i <$axis>
```

$ndlist	ND-list to modify.
$i	Index input, parsed with *::ndlist::ParseIndex*.
$axis	Axis to remove at. Default 0.

Example 33: Filtering a list by removing elements

Code:

```
set x [range 10]
puts [nremove $x [find $x > 4]]
```

Output:

```
0 1 2 3 4
```

Example 34: Deleting a column from a matrix

Code:

```
set a {{1 2 3} {4 5 6} {7 8 9}}
puts [nremove $a 2 1]
```

Output:

```
{1 2} {4 5} {7 8}
```

Insertion and Concatenation

The command *ninsert* inserts an ND-list into another ND-list at a specified index and axis. The ND-lists must agree in dimension at all other axes. If "end" or "end-integer" is used for the index, it will insert after the index. Otherwise, it will insert before the index. The command *ncat* is shorthand for inserting at "end", and concatenates two ND-lists.

```
ninsert $nd $ndlist1 $index $ndlist2 <$axis>
```

```
ncat $nd $ndlist1 $ndlist2 <$axis>
```

$nd	Rank of ND-list (e.g. 2D, 2d, or 2 for a matrix).
$ndlist1 $ndlist2	ND-lists to combine.
$index	Index to insert at.
$axis	Axis to insert/concatenate at (default 0).

Example 35: Inserting a column into a matrix

Code:

```
set matrix {{1 2} {3 4} {5 6}}
set column {A B C}
puts [ninsert 2D $matrix 1 $column 1]
```

Output:

```
{1 A 2} {3 B 4} {5 C 6}
```

Example 36: Concatenate tensors

Code:

```
set x [nreshape {1 2 3 4 5 6 7 8 9} 3 3 1]
set y [nreshape {A B C D E F G H I} 3 3 1]
puts [ncat 3D $x $y 2]
```

Output:

```
{{1 A} {2 B} {3 C}} {{4 D} {5 E} {6 F}} {{7 G} {8 H} {9 I}}
```

Changing Order of Axes

The command *nswapaxes* is a general purpose transposing function that swaps the axes of an ND-list. For simple matrix transposing, the command *transpose* can be used instead.

```
nswapaxes $ndlist $axis1 $axis2
```

$ndlist	ND-list to manipulate.
$axis1 $axis2	Axes to swap.

The command *nmoveaxis* moves a specified source axis to a target position. For example, moving axis 0 to position 2 would change "i,j,k" to "j,k,i".

```
nmoveaxis $ndlist $source $target
```

$ndlist	ND-list to manipulate.
$source	Source axis.
$target	Target position.

The command *npermute* is more general purpose, and defines a new order for the axes of an ND-list. For example, the axis list "1 0 2" would change "i,j,k" to "j,i,k".

```
npermute $ndlist $axis ...
```

$ndlist	ND-list to manipulate.
$axis ...	List of axes defining new order.

Example 37: Changing tensor axes

Code:

```
set x {{{1 2} {3 4}} {{5 6} {7 8}}}
set y [nswapaxes $x 0 2]
set z [nmoveaxis $x 0 2]
puts [lindex $x 0 0 1]
puts [lindex $y 1 0 0]
puts [lindex $z 0 1 0]
```

Output:

```
2
2
2
```

ND Functional Mapping

The command *napply* applies a command over each element of an ND-list, and returns the result. The commands *napply2* maps element-wise over two ND-lists. If the input lists have different shapes, they will be expanded to their maximum dimensions with *nexpand* (if compatible).

```
napply $nd $command $ndlist $arg ...
```

```
napply2 $nd $command $ndlist1 $ndlist2 $arg ...
```

$nd	Rank of ND-list (e.g. 2D, 2d, or 2 for a matrix).
$ndlist	ND-list to map over.
$ndlist1 $ndlist2	ND-lists to map over, element-wise.
$command	Command prefix to map with.
$arg ...	Additional arguments to append to command after ND-list element.

Example 38: Chained functional mapping over a matrix

Code:

```
napply 2D puts [napply 2D {format %.2f} [napply 2D expr {{1 2} {3 4}} + 1]]
```

Output:

```
2.00
3.00
4.00
5.00
```

Example 39: Format columns of a matrix

Code:

```
set data {{1 2 3} {4 5 6} {7 8 9}}
set formats {{%.1f %.2f %.3f}}
puts [napply2 2D format $formats $data]
```

Output:

```
{1.0 2.00 3.000} {4.0 5.00 6.000} {7.0 8.00 9.000}
```

Reducing an ND-list

The command *nreduce* combines *nmoveaxis* and *napply* to reduce an axis of an ND-list with a function that reduces a vector to a scalar, like *max* or *sum*.

```
nreduce $nd $command $ndlist <$axis> <$arg ...>
```

$nd	Rank of ND-list (e.g. 2D, 2d, or 2 for a matrix).
$command	Command prefix to map with.
$ndlist	ND-list to map over.
$axis	Axis to reduce. Default 0.
$arg ...	Additional arguments to append to command after ND-list elements.

Example 40: Matrix row and column statistics

Code:

```
set x {{1 2} {3 4} {5 6} {7 8}}
puts [nreduce 2D max $x]; # max of each column
puts [nreduce 2D max $x 1]; # max of each row
puts [nreduce 2D sum $x]; # sum of each column
puts [nreduce 2D sum $x 1]; # sum of each row
```

Output:

```
7 8
2 4 6 8
16 20
3 7 11 15
```

Generalized N-Dimensional Mapping

The command *nmap* is a general purpose mapping function for N-dimensional lists in Tcl, and the command *nexpr* a special case for math expressions. If multiple ND-lists are provided for iteration, they must be expandable to their maximum dimensions. The actual implementation flattens all the ND-lists and calls the Tcl *lmap* command, and then reshapes the result to the target dimensions. So, if "continue" or "break" are used in the map body, it will return an error.

```
nmap $nd $varName $ndlist <$varName $ndlist ...> $body
```

$nd	Rank of ND-list (e.g. 2D, 2d, or 2 for a matrix).
$varName	Variable name to iterate with.
$ndlist	ND-list to iterate over.
$body	Tcl script to evaluate at every loop iteration.

Example 41: Expand and map over matrices

Code:

```
set phrases [nmap 2D greeting {{hello goodbye}} subject {world moon} {
    list $greeting $subject
}]
napply 2D puts $phrases
```

Output:

```
hello world
goodbye world
hello moon
goodbye moon
```

Loop Index Access

The iteration indices of *nmap* can be accessed with the commands i, j, and k. The commands j and k are simply shorthand for i with axes 1 and 2.

```
i <$axis>
```

```
j
```

```
k
```

$axis	Dimension to access mapping index at. Default 0. If -1, returns the linear index of the loop.

Example 42: Finding index tuples that match criteria

Code:

```
set x {{1 2 3} {4 5 6} {7 8 9}}
set indices {}
nmap 2D xi $x {
    if {$xi > 4} {
        lappend indices [list [i] [j]]
    }
}
puts $indices
```

Output:

```
{1 1} {1 2} {2 0} {2 1} {2 2}
```

ND-Arrays

The command *narray* is a TclOO class based on the superclass *::vutil::ValueContainer*, from the package vutil. It is an object-oriented approach to array manipulation and processing.

```
narray new $nd $varName <$ndlist>
```

```
narray create $name $nd $varName <$ndlist>
```

$nd	Rank of ND-array (e.g. 2D, 2d, or 2 for a matrix).
$varName	Variable to store object name for access and garbage collection. Variable names are restricted to word characters and namespace delimiters only.
$ndlist	ND-list value to store in ND-array. Default blank.
$name	Name of object if using "create" method.

Value, Rank, Shape, and Size

The value is accessed by calling the object by itself, the rank is accessed with the method *rank*, and the shape and size are accessed with the methods *shape* and *size*.

```
$narrayObj rank
```

```
$narrayObj shape <$axis>
```

```
$narrayObj size
```

$axis	Axis to get dimension along. Default blank for all axes.

Example 43: Creating ND-arrays

Code:

```
# Create new ND-arrays
narray new 2D x {{1 2 3} {4 5 6} {7 8 9}}
narray new 1D y {hello world}
# Print rank and value of ND-arrays
puts "[$x rank], [$x]"
puts "[$y rank], [$y]"
```

Output:

```
2, {1 2 3} {4 5 6} {7 8 9}
1, hello world
```

Indexing

The "`@`" operator uses *nget* to access a portion of the ND-array.

```
$narrayObj @ $i ...
```

`$i ...`	Index inputs corresponding with rank of ND-array.

Example 44: Accessing portions of an ND-array

Code:

```
narray new 2D x {{1 2 3} {4 5 6} {7 8 9}}
puts [$x @ 0 2]
puts [$x @ 0:end-1 {0 2}]
```

Output:

```
3
{1 3} {4 6}
```

Copying

The operator "`-->`" copies the ND-array to a new variable, and returns the new object. If indices are specified, the new ND-array object will have the rank of the indexed range.

```
$narrayObj <@ $i ...> --> $varName
```

`$i ...`	Indices to access. Default all.
`$varName`	Variable to store object name for access and garbage collection. Variable names are restricted to word characters and namespace delimiters only.

Example 45: Copying a portion of an ND-array

Code:

```
narray new 2 x {{1 2 3} {4 5 6}}
$x @ 0* : --> y; # Row vector (flattened to 1D)
puts "[$y rank], [$y]"
```

Output:

```
1, 1 2 3
```

Evaluation/Mapping

The command *neval* maps over ND-arrays using *nmap*. The command *nexpr* is a special case that passes input through the Tcl *expr* command. ND-arrays can be referred to with "`$@ref`", where "ref" is the name of the ND-array variable. Portions of an ND-array can be mapped over with the notation "`$@ref($i,...)`". Input ND-arrays must all agree in rank or be scalar. Additionally, they must have compatible dimensions.

```
neval $body <$self> <$rankVar>
```

```
nexpr $expr <$self> <$rankVar>
```

`$body`	Script to evaluate, with "`$@ref`" notation for object references.
`$expr`	Expression to evaluate, with "`$@ref`" notation for object references.
`$self`	Object to refer to with "`$@`". Default blank.
`$rankVar`	Variable to store resulting rank in. Default blank.

Example 46: Get distance between elements in a vector

Code:

```
narray new 1D x {1 2 4 7 11 16}
puts [nexpr {$@x(1:end) - $@x(0:end-1)}]
```

Output:

```
1 2 3 4 5
```

Example 47: Outer product of two vectors

Code:

```
narray new 2D x {1 2 3}
narray new 2D y {{4 5 6}}
puts [nexpr {$@x * $@y}]
```

Output:

```
{4 5 6} {8 10 12} {12 15 18}
```

Modification

The assignment operator, "=", sets the value of the entire ND-array, or of a specified range. The math assignment operator, ":=", sets the value, passing the input through the *nexpr* command. Both assignment operators return the object.

```
$narrayObj <@ $i ...> = $value
```

```
$narrayObj <@ $i ...> := $expr
```

$i ...	Indices to modify. Default all.
$value	Value to assign. Blank to remove values.
$expr	Expression to evaluate.

If using the math assignment operator, the ND-array or indexed range can be accessed with the alias "$.", and the elements of the array or indexed range can be accessed with "$@".

```
$.  $arg ...
```

$arg ...	Method arguments for object.

Example 48: Element-wise modification of a vector

Code:

```
# Create blank vectors and assign values
[narray new 1D x] = {1 2 3}
[narray new 1D y] = {10 20 30}
# Add one to each element
puts [[$x := {$@ + 1}]]
# Double the last element
puts [[$x @ end := {$@ * 2}]]
# Element-wise addition of vectors
puts [[$x := {$@ + $@y}]]
```

Output:

```
2 3 4
2 3 8
12 23 38
```

Removal/Insertion

The method *remove* removes portions of an ND-array along a specified axis, returning the object.

The method *insert* inserts values into an ND-array at a specified index/axis, returning the object.

```
$narrayObj remove $i <$axis>
```

```
$narrayObj insert $i $sublist <$axis>
```

$i	Indices to remove/insert at.
$sublist	Value to insert.
$axis	Axis to remove/insert at (default 0).

Example 49: Removing elements from a vector

Code:

```
narray new 1 vector {1 2 3 4 5 6 7 8}
# Remove all odd numbers
$vector remove [find [nexpr {$@vector % 2}]]
puts [$vector]
```

Output:

```
2 4 6 8
```

Example 50: Inserting a column into a matrix

Code:

```
narray new 2 matrix {{1 2} {3 4} {5 6}}
$matrix insert 1 {A B C} 1
puts [$matrix]
```

Output:

```
{1 A 2} {3 B 4} {5 C 6}
```

Map/Reduce

The method *apply* maps a command over the ND-array, returning the value, and the method *reduce* applies a reducing command over a specified axis, returning the value. Both commands do not modify the object.

```
$narrayObj apply $command $arg ...
```

```
$narrayObj reduce $command <$axis> $arg ...
```

$command	Command prefix to map over the ND-list object.
$arg ...	Additional arguments to append to command.
$axis	Axis to reduce at (default 0).

Example 51: Map a command over a list

Code:

```
narray new 1 text {The quick brown fox jumps over the lazy dog}
puts [$text apply {string length}]; # Print the length of each word
```

Output:

```
3 5 5 3 5 4 3 4 3
```

Example 52: Get column statistics of a matrix

Code:

```
narray new 2 matrix {{1 2 3} {4 5 6} {7 8 9}}
# Convert to double-precision floating point
$matrix = [$matrix apply ::tcl::mathfunc::double]
# Get maximum and minimum of each column
puts [$matrix reduce max]
puts [$matrix reduce min]
```

Output:

```
7.0 8.0 9.0
1.0 2.0 3.0
```

Temporary Object Evaluation

The pipe operator, "|", copies the ND-array to a temporary object, and evaluates the method. Returns the result of the method, or the value of the temporary object. This operator is useful for converting methods that modify the object to methods that return a modified value.

```
$narrayObj <@ $i ...> | $method $arg ...
```

`$i ...`	Indices to access. Default all.
`$method`	Method to evaluate.
`$arg ...`	Arguments to pass to method.

Example 53: Temporary object value

Code:

```
# Create a matrix
narray new 2 x {{1 2 3} {4 5 6}}
# Print value with first row doubled.
puts [$x | @ 0* : := {$@ * 2}]
# Source object was not modified
puts [$x]
```

Output:

```
{2 4 6} {4 5 6}
{1 2 3} {4 5 6}
```

Reference Variable Evaluation

The operator "`&`" copies the ND-array value or range to a reference variable, and evaluates a body of script. The changes made to the reference variable will be applied to the object, and if the variable is unset, the object will be deleted. If no indices are specified and the variable is unset in the script, the ND-array object will be destroyed. Returns the result of the script.

```
$narrayObj <@ $i ...> & $refName $body
```

`$i ...`	Indices to access. Default all.
`$refName`	Variable name to use for reference.
`$body`	Body to evaluate.

Example 54: Appending a vector

Code:

```
# Create a 1D list
narray new 1 x {1 2 3}
# Append the list
$x & ref {lappend ref 4 5 6}
puts [$x]
# Append a subset of the list
$x @ end* & ref {lappend ref 7 8 9}
puts [$x]
```

Output:

```
1 2 3 4 5 6
1 2 3 4 5 {6 7 8 9}
```

Command Index