N-Dimensional Lists (ndlist)

Version 0.13

Alex Baker

https://github.com/ambaker1/ndlist

June 8, 2025

Abstract

In Tcl, everything is a string, and most things are lists. This package expands on this, by proposing a new paradigm for Tcl: everything is an ND-list. An ND-list is an arbitrary-rank tensor containing any valid Tcl value. It is implemented by using nested Tcl lists, and this package provides utilities for ND-list access, modification, manipulation, and element-wise operations.

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 2.1
tin autoadd 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 *where* 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.

```
where $list <$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 [where $x > 2]]
```

Output:

```
2.3 4.0 2.5 5.6
```

Linear Interpolation

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

```
linsteps $step $targets
```

$step	Maximum step size
$targets	List of 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
```

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 13: 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 14: 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 15: 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 *eye* generates an identity matrix.

```
eye $n
```

`$n`	Size of identity matrix

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 16: 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 17: 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 18: 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)

All Tcl values are ND-lists. An 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 0D-lists, which are simply strings that either have no list representation or are of list length 1. 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". In general, if a value is a valid for N dimensions, it will also be valid for dimensions 0 to N-1.

The command *ndims* returns the rank of an ND-list, and the command *ndims_multiple* returns the rank that is compatible with multiple ND-lists. By default, it automatically determines the rank for the data, but if a rank is provided, it will validate that the ND-list or ND-lists are compatible with the provided rank.

```
ndims $ndlist <$rank>
```

```
ndims_multiple $ndlists <$rank>
```

$ndlist	ND-list.
$ndlists	List of ND-lists.
$rank	Rank of ND-list (e.g. 2 for matrix) or "auto" for auto-rank. Default "auto".

Example 19: Rank of an ND-list

Code:

```
set x {1}
set y {1 2 {hello world}}; # note that this is not a valid 2D list
set z {{1 2 3} {4 5 6}}
puts [ndims $x]; # 0
puts [ndims $y]; # 1
puts [ndims $z]; # 2
# the only rank that works for x, y, and z is 1
puts [ndims_multiple [list $x $y $z]]; # 1
```

Output:

```
0
1
2
1
```

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 $ndlist <$rank>
```

```
nsize $ndlist <$rank>
```

$ndlist	ND-list to get shape/size of.
$rank	Rank of ND-list (e.g. 2 for matrix) or "auto" for auto-rank. Default "auto".

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

Code:

```
# Create a 3D list
set x {{{1 2} {3 4} {5 6}} {{7 8} {9 10} {11 12}}}
# Get the shape and size for different rank interpretations
puts [list [nshape $x] [nsize $x]]; # auto-rank (3)
puts [list [nshape $x 1] [nsize $x 1]]; # rank 1
puts [list [nshape $x 2] [nsize $x 2]]; # rank 2
puts [list [nshape $x 3] [nsize $x 3]]; # rank 3
puts [list [nshape $x 4] [nsize $x 4]]; # rank 4
```

Output:

```
{2 3 2} 12
2 2
{2 3} 6
{2 3 2} 12
{2 3 2 1} 12
```

Initialization

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

```
nfull $value $shape
```

$value	Value to repeat
$shape	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 $shape
```

$shape	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 $repeats
```

$value	Value to repeat
$repeats	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 $shape
```

$ndlist	ND-list to expand.
$shape	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 $pads
```

$ndlist	ND-list to pad.
$value	Value to pad with.
$pads	Number of elements to pad along each axis. Negative to prepend.

Example 25: Padding an ND-list with a value

Code:

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

Output:

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

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

```
nextend $ndlist $value $shape
```

$ndlist	ND-list to extend.
$value	Value to pad with.
$shape	New shape of ND-list. To keep the shape at an axis, use -1.

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}]; # -1 preserves size at axis 0
```

Output:

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

Flattening and Reshaping

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

```
nflatten $ndlist <$rank>
```

$ndlist	ND-list to flatten.
$rank	Rank of ND-list (e.g. 2 for matrix) or "auto" for auto-rank. Default "auto".

Example 27: Flattening a 3D tensor

Code:

```
puts [nflatten {{{1 2} {3 4}} {{5 6} {7 8}}}]
```

Output:

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

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

```
nreshape $vector $shape
```

$vector	Vector (1D-list) to reshape.
$shape	Shape of ND-list. One axis may be dynamic, denoted with -1.

Example 28: Reshape a vector to a matrix with three columns

Code:

```
puts [nreshape {1 2 3 4 5 6} {-1 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
# TIP: You can use the Tcl 'interp alias' command to create a convenient shortcut to nget
interp alias {} @ {} nget
puts [@ $A 0 :]; # get row matrix
```

Output:

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

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.

If using the "`= $expr`" notation, it will call *nexpr*, where "self" is the ND-list or indexed range being modified and "rank" is "auto" if no index arguments are provided, otherwise it is equal to the number of index arguments minus the number of single index (S) arguments (see *::ndlist::ParseIndex*).

```
nset $varName $i ...  ($sublist | = $expr)
```

```
nreplace $ndlist $i ...  ($sublist | = $expr)
```

`$varName`	Variable that contains an ND-list.
`$ndlist`	ND-list to modify.
`$i ...`	Index inputs, parsed with *::ndlist::ParseIndex*.
`$sublist`	Replacement list, or blank to delete values.
`$expr`	Expression to evaluate and replace values with. Indexed range can be accessed with "`@.`" for convenience.

Example 31: ND-list modification

Code:

```
# Swap rows in a matrix (by reference)
set a {{1 2 3} {4 5 6} {7 8 9}}
nset a {1 0} : [nget $a {0 1} :]
puts $a
# Element-wise operation on portion of vector (by value)
set b [range 10]
puts [nreplace $b 0:2:end = {@. + 10}]
```

Output:

```
{4 5 6} {1 2 3} {7 8 9}
10 1 12 3 14 5 16 7 18 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 32: Filtering a list by removing elements

Code:

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

Output:

```
0 1 2 3 4
```

Example 33: 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 $ndlist1 $index $ndlist2 <$axis> <$rank>
```

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

$ndlist1 $ndlist2	ND-lists to combine.
$index	Index to insert at.
$axis	Axis to insert/concatenate at (default 0).
$rank	Rank of ND-list (e.g. 2 for matrix) or "auto" for auto-rank. Default "auto".

Example 34: Inserting a column into a matrix

Code:

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

Output:

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

Example 35: 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 $x $y 2 3]
```

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 $order
```

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

Example 36: Changing tensor axes

Code:

```
set x {{{1 2} {3 4}} {{5 6} {7 8}}}; # 3D tensor
set y [nswapaxes $x 0 2]; # k,j,i
set z [nmoveaxis $x 0 2]; # j,k,i
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 $command $ndlist <$suffix> <$rank>
```

```
napply2 $command $ndlist1 $ndlist2 <$suffix> <$rank>
```

$ndlist	ND-list to map over.
$ndlist1 $ndlist2	ND-lists to map over, element-wise.
$command	Command prefix to map with.
$suffix	Additional arguments to append after ND-list elements. Default blank.
$rank	Rank of ND-list (e.g. 2 for matrix) or "auto" for auto-rank. Default "auto".

Example 37: Chained functional mapping over a matrix

Code:

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

Output:

```
2.00
3.00
4.00
5.00
```

Example 38: Format columns of a matrix

Code:

```
set data {{1 2 3} {4 5 6} {7 8 9}}
set formats {{%.1f %.2f %.3f}}
puts [napply2 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 $command $ndlist <$axis> <$suffix> <$rank>
```

$command	Command prefix to map with.
$ndlist	ND-list to map over.
$axis	Axis to reduce. Default 0.
$suffix	Additional arguments to append after ND-list elements. Default blank.
$rank	Rank of ND-list (e.g. 2 for matrix) or "auto" for auto-rank. Default "auto".

Example 39: Matrix row and column statistics

Code:

```
set x {{1 2} {3 4} {5 6} {7 8}}
puts [nreduce max $x]; # max of each column
puts [nreduce max $x 1]; # max of each row
puts [nreduce sum $x]; # sum of each column
puts [nreduce 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. 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. If rank is not specified, it will automatically determine the rank using *ndims_multiple* and the referenced ND-lists.

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

$rank	Rank (e.g. 2 for matrix) or "auto" for auto-rank. Default "auto".
$varName	Variable name to iterate with.
$ndlist	ND-list to iterate over.
$body	Tcl script to evaluate at every loop iteration.

Example 40: Expand and map over matrices

Code:

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

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 41: Finding index tuples that match criteria

Code:

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

Output:

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

Element-Wise Expressions

The command *neval* maps over ND-arrays using *nmap*, but without the need to specify looping variables, and the command *nexpr* is a special case that passes input through the Tcl *expr* command. ND-lists can be referred to with "`@ref`", where "ref" is the name of the variable storing the ND-list. If an ND-list is provided as "self", it can be referred to with "`@.`" for convenience. Additionally, portions of an ND-list can be mapped over with the notation "`@ref($i,...)`", where "`$i,...`" are raw index arguments (does not do any substitution). If rank is not specified, it will automatically determine the rank using *ndims_multiple* and the referenced/indexed ND-lists.

```
neval $body <$self> <$rank>
```

```
nexpr $expr <$self> <$rank>
```

`$body`	Script to evaluate, with "`@ref`" notation for ND-list references.
`$expr`	Expression to evaluate, with "`@ref`" notation for ND-list references.
`$self`	ND-list to refer to with "`@.`". Default blank.
`$rank`	Rank (e.g. 2 for matrix) or "auto" for auto-rank. Default "auto".

Example 42: Element-wise operations

Code:

```
set x {1 2 3}; # vector, length 3
set y {{4 5 6}}; # matrix, shape {1 3}
set z {1 2 4 7 11 16}; # vector, length 6
puts [nexpr {@x * @y}]; # outer product of two vectors (creates matrix)
puts [nexpr {@x + @z}]; # expands vector x to match length of z
puts [nexpr {@z(1:end) - @z(0:end-1)}]; # distance between vector elements
puts [nexpr {@. * 2.0} {4 3 8}]; # self-operation using @. notation
```

Output:

```
{4 5 6} {8 10 12} {12 15 18}
2 4 7 8 13 19
1 2 3 4 5
8.0 6.0 16.0
```

Prefix Notation Math

Tcl is unique in that everything is a command. So, that means that it does not natively support the typical algebraic "infix" notation that people are used to. Instead, the Tcl *expr* command provides a sub-language that parses scalar math in the familiar "infix" notation. The ndlist *nexpr* command was designe to extend the functionality of *expr* to provide support for element-wise operations. However, in code that is heavy in math, having a lot of "expr" or "nexpr" calls can make code difficult to read. There has been a lot of discussion in the Tcl community about this problem, and the general consensus is that it was addressed in Tcl TIP #174, which added the ::tcl::mathop namespace to the core. This change to Tcl provided users with easy access to math operators as commands. The ndlist package expands upon this by providing equivalent element-wise operators as commands, with the exception that it only provides element-wise math operators. This is because the ndlist math operator commands do not have an option to specify the rank of the input lists, which could lead to unexpected results for some string/list operations.

The available element-wise operators are ~, `!`, `-`, `+`, `*`, `<<`, `**`, `/`, `%`, `>>`, `&`, `|`, `^`, `==`, `!=`, `<`, `<=`, `>`, and `>=`.

Full documentation of these operators can be found in the Tcl ::tcl::mathop namespace documentation.

```
.$op $ndlist ...
```

`$op`	Tcl numeric math operator.
`$ndlist ...`	Numeric ND-lists to apply operator to.

Example 43: Prefix notation math

Code:

```
namespace path {::tcl::mathfunc ::tcl::mathop}; # exposes 'expr' functions and operators
# Scalar math
set x [+ 1 2]; # 3
puts [** $x 2]
# Element-wise math
set x [.+ {10 20} [nreshape [range 6] {2 3}]]; # {10 11 12} {23 24 25}
puts [napply double [.- [nget $x 0* 0:end-1]]]
```

Output:

```
9
-10.0 -11.0
```

File Import/Export

The commands *readFile* and *writeFile* perform simple data import/export, while the commands *readMatrix* and *writeMatrix* dynamically convert files to matrix format and matrices to the specified file format (e.g. file extension .csv will call conversion functions *mat2csv* and *csv2mat*).

```
readFile <$option $value ...> <-newline> $file
```

```
readMatrix <$option $value ...> <-newline> $file
```

`$option $value ...`	File configuration options, see Tcl *fconfigure* command.
`-newline`	Option to read the final newline if it exists.
`$file`	File to read data from.

```
writeFile <$option $value ...> <-nonewline> $file $data
```

```
writeMatrix <$option $value ...> <-nonewline> $file $data
```

`$option $value ...`	File configuration options, see Tcl *fconfigure* command.
`-nonewline`	Option to not write a final newline.
`$file`	File to write data to.
`$data`	Data to write to file.

Example 44: File import/export

Code:

```
# Export matrix to file (converts to csv)
writeMatrix example.csv {{foo bar} {hello world}}
# Read CSV file
puts [readFile example.csv]
puts [readMatrix example.csv]; # converts from csv to matrix
file delete example.csv
```

Output:

```
foo,bar
hello,world
{foo bar} {hello world}
```

Data Conversions

The commands *mat2txt* and *txt2mat* convert between matrix and space-delimited text, where new-lines separate rows. Escaping of spaces and newlines is consistent with Tcl rules for valid lists.

```
mat2txt $mat
```

```
txt2mat $txt
```

$mat	Matrix value.
$txt	Space-delimited values.

The commands *mat2csv* and *csv2mat* convert between matrix and CSV-formatted text, where new lines separate rows. Commas and newlines are escaped with quotes, and quotes are escaped with double-quotes.

```
mat2csv $mat
```

```
csv2mat $csv
```

$mat	Matrix value.
$csv	Comma-separated values.

Example 45: Data conversions

Code:

```
set matrix {{A B C} {{hello world} foo,bar {"hi"}}}
puts {TXT format:}
puts [mat2txt $matrix]
puts {CSV format:}
puts [mat2csv $matrix]
```

Output:

```
TXT format:
A B C
{hello world} foo,bar {"hi"}
CSV format:
A,B,C
hello world,"foo,bar","""hi"""
```

Command Index