N-Dimensional Lists (ndlist)

Version 0.8

Alex Baker

https://github.com/ambaker1/ndlist

April 27, 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 "*float*".

```
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 "*float*"

```
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 "*float*".

```
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. Basic math operators can be mapped over a list with the command *lop*.

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

```
lop $list $op $arg...
```

$list	List to map over.
$command	Command prefix to map with.
$op	Math operator (see ::tcl::mathop documentation).
$arg ...	Additional arguments to append to command after each list element.

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
```

Mapping Over Two Lists

The commands *lapply* and *lop* only map over one list. The commands *lapply2* and *lop2* allow you to map, element-wise, over two lists. List lengths must be equal.

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

```
lop2 $list1 $op $list2 $arg...
```

`$list1 $list2`	Lists to map over, element-wise.
`$command`	Command prefix to map with.
`$op`	Math operator (see ::tcl::mathop documentation).
`$arg ...`	Additional arguments to append to command after list elements.

Example 9: Mapping over two lists

Code:

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

Output:

```
hello world
goodbye moon
```

Example 10: Adding two lists together

Code:

```
puts [lop2 {1 2 3} + {2 3 2}]
```

Output:

```
3 5 5
```

List Math

The Tcl command *lmap* allows you to loop over an arbitrary number of lists in parallel, evaluating a script at each iteration, and collecting the results of each loop iteration into a new list. The command *lexpr* is an extension of this concept, just calling *lmap* and passing the input through the Tcl *expr* command.

```
lexpr $varList $list <$varList $list ...> $expr
```

`$varList ...`	List(s) of variables to iterate with.
`$list ...`	List(s) to iterate over.
`$expr`	Tcl expression to evaluate at every loop iteration.

Example 11: Filtering a list

Code:

```
set numbers [range 10]
set odds [lexpr x $numbers {$x % 2 ? $x : [continue]}]; # only odd numbers
puts $odds
```

Output:

```
1 3 5 7 9
```

Example 12: Adding three lists together

Code:

```
set x {1 2 3}
set y {2 9 2}
set z {5 -2 0}
puts [lexpr xi $x yi $y zi $z {$xi + $yi + $zi}]
```

Output:

```
8 9 5
```

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 13: 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.

Example 14: 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
```

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 15: Normalizing a vector

Code:

```
set x {3 4}
set x [lop $x / [norm $x]]
puts $x
```

Output:

```
0.6 0.8
```

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 16: 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 17: 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 18: 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 19: 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 20: 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 21: 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 22: 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 23: 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 dimensions of.
$axis	Axis to get dimension along. Blank for all.

Example 24: Getting shape and size of an ND-list

Code:

```
set A [ndlist 2D {{1 2 3} {4 5 6}}]
puts [nshape 2D $A]
puts [nsize 2D $A]
```

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 25: 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 26: 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 27: 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 28: 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 29: 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 30: 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 *nreshape* reshapes a vector into a compatible shape. Vector length must equal target ND-list size.

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

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

Example 31: Reshape a vector to a matrix

Code:

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

Output:

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

The inverse is *nflatten*, which flattens an ND-list to a vector, which can be then used with *nreshape*.

```
nflatten $nd $ndlist
```

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

Example 32: 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}}
```

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 33: 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*. The number of index arguments determines the interpreted dimensions.

Example 34: 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*. The number of index inputs determines the interpreted dimensions.
$sublist	Replacement list, or blank to delete values.

Example 35: 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 36: 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 $nd $ndlist $i <$axis>
```

$nd	Rank of ND-list (e.g. 2D, 2d, or 2 for a matrix).
$ndlist	ND-list to modify.
$i	Index input, parsed with *::ndlist::ParseIndex*.
$axis	Axis to remove at. Default 0.

Example 37: Filtering a list by removing elements

Code:

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

Output:

```
0 1 2 3 4
```

Example 38: 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}
```

Appending

The command *nappend* is a generalized append for Tcl. For 0D, it just calls the Tcl *append* command. For 1D, it just calls the Tcl *lappend* command. For ND, it verifies that the (N-1)D inputs have the same shape as the elements of the ND-list, and then calls the Tcl *lappend* command, appending along axis 0. For example, for 2D, it verifies that the list lengths match the number of columns of the matrix.

```
nappend $nd $varName $arg ...
```

`$nd`	Rank of ND-list (e.g. 2D, 2d, or 2 for a matrix).
`$varName`	Variable that contains an ND-list.
`$arg ...`	(N-1)D lists (or strings for 0D) to append to ND-list.

Example 39: Scalar and list append

Code:

```
set a {}
nappend 0D a foo
nappend 0D a bar
nappend 1D a {hello world}
puts $a
```

Output:

```
foobar {hello world}
```

Example 40: Adding rows to a matrix (checks dimensions)

Code:

```
set a {}
nappend 2D a {1 2 3}
nappend 2D a {4 5 6}
nappend 2D a {7 8 9}
puts $a
```

Output:

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

Insertion and Concatenation

The command *ninsert* allows you to insert an ND-list into another ND-list at a specified index and axis, as long as the ND-lists 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 41: 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 42: 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 43: 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* simply applies a command over each element of an ND-list, and returns the result. Basic math operators can be mapped over an ND-list with the command *nop*, which is a special case of *napply*, using the ::tcl::mathop namespace.

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

```
nop $nd $ndlist $op $arg...
```

$nd	Rank of ND-list (e.g. 2D, 2d, or 2 for a matrix).
$ndlist	ND-list to map over.
$command	Command prefix to map with.
$op	Math operator (see ::tcl::mathop documentation).
$arg ...	Additional arguments to append to command after ND-list element.

Example 44: 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 45: Element-wise operations

Code:

```
puts [nop 1D {1 2 3} + 1]
puts [nop 2D {{1 2 3} {4 5 6}} > 2]
```

Output:

```
2 3 4
{0 0 1} {1 1 1}
```

Mapping Over Two ND-lists

The commands *napply* and *nop* only map over one ND-list. The commands *napply2* and *nop2* allow you to map, 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).

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

```
nop2 $nd $ndlist1 $op $ndlist2 $arg...
```

$nd	Rank of ND-list (e.g. 2D, 2d, or 2 for a matrix).
$ndlist1 $ndlist2	ND-lists to map over, element-wise.
$command	Command prefix to map with.
$op	Math operator (see ::tcl::mathop documentation).
$arg ...	Additional arguments to append to command after ND-list elements.

Example 46: 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}
```

Example 47: Adding matrices together

Code:

```
set A {{1 2} {3 4}}
set B {{4 9} {3 1}}
puts [nop2 2D $A + $B]
```

Output:

```
{5 11} {6 5}
```

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 48: 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
```

```
nexpr $nd $varName $ndlist <$varName $ndlist ...> $expr
```

$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.
$expr	Tcl expression to evaluate at every loop iteration.

Example 49: 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
```

Example 50: Adding two matrices together, element-wise

Code:

```
set x {{1 2} {3 4}}
set y {{4 1} {3 9}}
set z [nexpr 2D xi $x yi $y {$xi + $yi}]
puts $z
```

Output:

```
{5 3} {6 13}
```

Generalized N-Dimensional Looping

The command *nforeach* is simply a version of *nmap* that returns nothing.

```
nforeach $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.

Loop Index Access

The iteration indices of *nmap*, *nexpr*, or *nforeach* 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 51: Finding index tuples that match criteria

Code:

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

Output:

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

Tabular Data Structure

This package provides an object-oriented tabular datatype in Tcl, building upon the type system framework provided by the vutil package.

The string representation of this datatype is a dictionary, with keys representing the table header, and values representing the table columns. The values in the first column must be unique, and are called the table "keys". Correspondingly, the first header entry is called the "keyname". The remaining header entries are called the table "fields", and the remaining columns are the data stored in the table. The conceptual layout of the table is illustrated in the figure below.

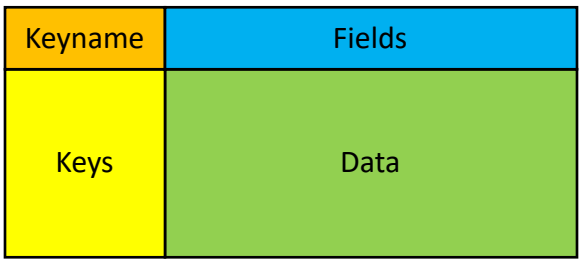

Figure 1: Conceptual Representation of Tabular Data Structure

There is no restriction on the type of data that can be stored in a table, as shown in the example below, which has keyname "key", keys "1 2 3" and fields "A B". Note that missing values are represented by blanks.

Example 52: String representation of tabular data type

Code:

```
key {1 2 3} A {5.6 {} 2.22} B {{hello world} 4.5 foo}
```

Creating Table Objects

This package provides the "table" type class, using the type system provided by the "vutil" package. The command *table* creates a new table object variable, which uses the class `::taboo::tblobj`.

```
table $refName <$value>
```

$refName	Reference name to tie object to.
$value	Value of table. Default blank.

Below is the example table used in the remainder of the documentation examples. Note that this format is also compatible with the string representation of Tcl arrays and dictionaries.

Example 53: Example table

Code:

```
table tableObj {
    key {1 2 3 4 5}
    x {3.44 4.61 8.25 5.20 3.26}
    y {7.11 1.81 7.56 6.78 9.92}
    z {8.67 7.63 3.84 1.11 4.56}
}
puts [$tableObj]
```

Output:

```
key {1 2 3 4 5} x {3.44 4.61 8.25 5.20 3.26} y {7.11 1.81 7.56 6.78 9.92} z {8.67 7.63 3.84
    1.11 4.56}
```

Wiping, Clearing, and Cleaning a Table

The method *wipe* removes all data from a table object, so that its state is the same as a fresh table. The method *clear* only removes the data and keys stored in the table, keeping the fields and other metadata. The method *clean* only removes keys and fields that have no data.

```
$tableObj wipe
```

```
$tableObj clear
```

```
$tableObj clean
```

Standard Methods

Because the table objects are object variables, they have the same basic methods provided by the "vutil" package. For more info on these methods, see the documentation for the "vutil" package.

```
$tableObj --> $refName
$tableObj <- $object
$tableObj = $value
$tableObj ::= $body
$tableObj info <$field>
$tableObj print <-nonewline> <$channelID>
$tableObj destroy
```

$refName	Reference name to copy to.
$object	Table object.
$value	Table value to assign.
$body	Tcl script to evaluate and set as table value.
$field	Field to query (fields "height" and "width" added).
$channelID	Open channel to print to.

Note that the methods `.=` and `:=` are also available, but they are not recommended for tables. Also note that "taboo" tables are always initialized, so the "exists" field of the object variable will always be true.

Example 54: Copying a table

Code:

```
$tableObj --> tableCopy
puts [$tableCopy info]
```

Output:

```
exists 1 height 5 type table value {key {1 2 3 4 5} x {3.44 4.61 8.25 5.20 3.26} y {7.11
    1.81 7.56 6.78 9.92} z {8.67 7.63 3.84 1.11 4.56}} width 3
```

Get/Set Keyname

The keyname of a table can be accessed or modified directly with their respective methods.

```
$tableObj keyname <$keyname>
```

$keyname	Header for table keys. Default blank to return current name.

Get Keys and Fields

The table keys and fields are ordered lists of the row and column names of the table. They can be queried with the methods *keys* and *fields*, respectively.

```
$tableObj keys <$index>
```

```
$tableObj fields <$index>
```

$index	Index arguments, using "ndlist" index notation. Default ":" for all keys/fields.

Table Dimensions

The number of keys can be queried with *$tableObj height* and the number of fields can be queried with *$tableObj width*. These are also properties accessible with the standard method *info*. Note that rows and columns with missing data will be counted.

```
$tableObj height
```

```
$tableObj width
```

Example 55: Accessing table keys and table dimensions

Code:

```
puts [$tableObj keys]
puts [$tableObj keys 0:end-1]
puts [$tableObj height]
```

Output:

```
1 2 3 4 5
1 2 3 4
5
```

Get Table Data (Dictionary Form)

The method *data* returns the table data in unsorted dictionary form, where blanks are represented by missing dictionary entries.

```
$tableObj data <$key>
```

$key	Key to get row dictionary from (default returns all rows).

Get Table Data (Matrix Form)

The method *values* returns a matrix (list of rows) that represents the data in the table, where the rows correspond to the keys and the columns correspond to the fields. Missing entries are represented by blanks in the matrix unless specified otherwise.

```
$tableObj values <$filler>
```

$filler	Filler for missing values, default blank.

Example 56: Getting table data in dictionary and matrix form

Code:

```
puts [$tableObj data]
puts [$tableObj data 3]
puts [$tableObj values]
```

Output:

```
1 {x 3.44 y 7.11 z 8.67} 2 {x 4.61 y 1.81 z 7.63} 3 {x 8.25 y 7.56 z 3.84} 4 {x 5.20 y 6.78
    z 1.11} 5 {x 3.26 y 9.92 z 4.56}
x 8.25 y 7.56 z 3.84
{3.44 7.11 8.67} {4.61 1.81 7.63} {8.25 7.56 3.84} {5.20 6.78 1.11} {3.26 9.92 4.56}
```

Check Existence of Table Keys/Fields

The existence of a table key, field, or table value can be queried with the method *exists*.

```
$tableObj exists key $key
$tableObj exists field $field
$tableObj exists value $key $field
```

`$key`	Key to check.
`$field`	Field to check.

Get Row/Column Indices

The row or column index of a table key or field can be queried with the method *find*. If the key or field does not exist, returns an error.

```
$tableObj find key $key
$tableObj find field $field
```

`$key`	Key to find.
`$field`	Field to find.

Example 57: Find column index of a field

Code:

```
puts [$tableObj exists field z]
puts [$tableObj find field z]
```

Output:

```
1
2
```

Table Entry and Access

Data entry and access to a table object can be done with single values with the methods *set* and *get*, entire rows with *rset* and *rget*, entire columns with *cset* and *cget*, or in matrix fashion with *mset* and *mget*. If entry keys/fields do not exist, they are added to the table. Additionally, since blank values represent missing data, setting a value to blank effectively unsets the table entry, but does not remove the key or field.

Single Value Entry and Access

The methods *set* and *get* allow for easy entry and access of single values in the table. Note that multiple field-value pairings can be used in *$tableObj set*.

```
$tableObj set $key $field $value ...
```

```
$tableObj get $key $field <$filler>
```

$key	Key of row to set/get data in/from.
$field	Field of column to set/get data in/from.
$value	Value to set.
$filler	Filler to return if value is missing. Default blank.

Example 58: Setting multiple values

Code:

```
$tableObj --> tableCopy
$tableCopy set 1 x 2.00 y 5.00 foo bar
puts [$tableCopy data 1]
```

Output:

```
x 2.00 y 5.00 z 8.67 foo bar
```

Row Entry and Access

The methods *rset* and *rget* allow for easy row entry and access. Entry list length must match table width or be scalar. If entry list is blank, it will delete the row, but not the key.

```
$tableObj rset $key $row
```

```
$tableObj rget $key <$filler>
```

$key	Key of row to set/get.
$row	List of values (or scalar) to set.
$filler	Filler for missing values. Default blank.

Column Entry and Access

The methods *cset* and *cget* allow for easy column entry and access. Entry list length must match table height or be scalar. If entry list is blank, it will delete the column, but not the field.

```
$tableObj cset $field $column
```

```
$tableObj cget $field <$filler>
```

$field	Field of column to set/get.
$column	List of values (or scalar) to set.
$filler	Filler for missing values. Default blank.

Matrix Entry and Access

The methods *mset* and *mget* allow for easy matrix-style entry and access. Entry matrix size must match table size or be scalar.

```
$tableObj mset $keys $fields $matrix
```

```
$tableObj mget $keys $fields <$filler>
```

$keys	List of keys to set/get (default all keys).
$fields	List of keys to set/get (default all keys).
$matrix	Matrix of values (or scalar) to set.
$filler	Filler for missing values. Default blank.

Below is an example of how you can construct a table from scratch. Note also how you can create a table using the "vutil" command *new* instead of the command *table*.

Example 59: Matrix entry and access

Code:

```
::vutil::new table T
$T add keys 1 2 3 4
$T add fields A B
$T mset [$T keys] [$T fields] 0.0; # Initialize as zero
$T mset [$T keys 0:2] A {1.0 2.0 3.0}; # Set subset of table
puts [$T values]
```

Output:

```
{1.0 0.0} {2.0 0.0} {3.0 0.0} {0.0 0.0}
```

Iterating Over Table Data

Table data can be looped through, row-wise, with the method *with*. Variables representing the key values and fields will be assigned their corresponding values, with blanks representing missing data. The variable representing the key (table keyname) is static, but changes made to field variables are reflected in the table. Unsetting a field variable or setting its value to blank unsets the corresponding data in the table.

```
$tableObj with $body
```

$body	Code to execute.

Example 60: Iterating over a table, accessing and modifying field values

Code:

```
$tableObj --> tableCopy
set a 20.0
$tableCopy add fields q
$tableCopy with {
    puts [list $key $x]; # access key and field value
    set q [expr {$x*2 + $a}]; # modify field value
}
puts [$tableCopy cget q]
```

Output:

```
1 3.44
2 4.61
3 8.25
4 5.20
5 3.26
26.88 29.22 36.5 30.4 26.52
```

Note: Just like in *dict with*, the key variable and field variables in *$tableObj with* persist after the loop.

Field Expressions

The method *expr* computes a list of values according to a field expression. In the same style as referring to variables with the dollar sign ($), the "at" symbol (@) is used by *$tableObj expr* to refer to field values, or row keys if the keyname is used. If any referenced fields have missing values for a table row, the corresponding result will be blank as well. The resulting list corresponds to the keys in the table.

```
$tableObj expr $fieldExpr
```

$fieldExpr	Field expression.

Editing Table Fields

Field expressions can be used to edit existing fields or add new fields in a table with the method *fedit*. If any of the referenced fields are blank, the corresponding entry will be blank as well.

```
$tableObj fedit $field $fieldExpr
```

$field	Field to set.
$fieldExpr	Field expression.

Example 61: Using field expressions

Code:

```
$tableObj --> tableCopy
set a 20.0
puts [$tableCopy cget x]
puts [$tableCopy expr {@x*2 + $a}]
$tableCopy fedit q {@x*2 + $a}
puts [$tableCopy cget q]
```

Output:

```
3.44 4.61 8.25 5.20 3.26
26.88 29.22 36.5 30.4 26.52
26.88 29.22 36.5 30.4 26.52
```

Querying Keys that Match Criteria

The method *filter* returns the keys in a table that match criteria in a field expression.

```
$tableObj query $fieldExpr
```

$fieldExpr	Field expression that results in boolean value (true or false, 1 or 0).

Example 62: Getting keys that match criteria

Code:

```
puts [$tableObj query {@x > 3.0 && @y > 7.0}]
```

Output:

```
1 3 5
```

Filtering Table Based on Criteria

The method *filter* filters a table to the keys matching criteria in a field expression.

```
$tableObj filter $fieldExpr
```

$fieldExpr	Field expression that results in boolean value (true or false, 1 or 0).

Example 63: Filtering table to only include keys that match criteria

Code:

```
$tableObj --> tableCopy
$tableCopy filter {@x > 3.0 && @y > 7.0}
puts [$tableCopy keys]
```

Output:

```
1 3 5
```

Searching a Table

Besides searching for specific field expression criteria with *$tableObj query*, keys matching criteria can be found with the method *search*. The method *search* searches a table using the Tcl *lsearch* command on the keys or field values. The default search method uses glob pattern matching, and returns matching keys. This search behavior can be changed with the various options, which are taken directly from the Tcl *lsearch* command. Therefore, while brief descriptions of the options are provided here, they are explained more in depth in the Tcl documentation, with the exception of the -inline option. The -inline option filters a table based on the search criteria.

```
$tableObj search <$option ...> <$field> $value
```

$option ...	Searching options. Valid options:
-exact	Compare strings exactly
-glob	Use glob-style pattern matching (default)
-regexp	Use regular expression matching
-sorted	Assume elements are in sorted order
-all	Get all matches, rather than the first match
-not	Negate the match(es)
-ascii	Use string comparison (default)
-dictionary	Use dictionary-style comparison
-integer	Use integer comparison
-real	Use floating-point comparison
-nocase	Search in a case-insensitive manner
-increasing	Assume increasing order (default)
-decreasing	Assume decreasing order
-bisect	Perform inexact match
-inline	Filter table instead of returning keys.
--	Signals end of options
$field	Field to search. If blank, searches keys.
$value	Value or pattern to search for

Note: If a field contains missing values, they will only be included in the search if the search options allow (e.g. blanks are included for string matching, but not for numerical matching).

Sorting a Table

The method *sort* sorts a table by keys or field values. The default sorting method is in increasing order, using string comparison. This sorting behavior can be changed with the various options, which are taken directly from the Tcl *lsort* command. Therefore, while brief descriptions of the options are provided here, they are explained more in depth in the Tcl documentation. Note: If a field contains missing values, the missing values will be last, regardless of sorting options.

```
$tableObj sort <$option ...> <$field ...>
```

$option ...	Sorting options. Valid options:
-ascii	Use string comparison (default)
-dictionary	Use dictionary-style comparison
-integer	Use integer comparison
-real	Use floating comparison
-increasing	Sort the list in increasing order (default)
-decreasing	Sort the list in decreasing order
-nocase	Compare in a case-insensitive manner
--	Signals end of options
$field ...	Fields to sort by (in order of sorting). If blank, sorts by keys.

Example 64: Searching and sorting

Code:

```
$tableObj --> tableCopy
puts [$tableCopy search -real x 8.25]; # returns first matching key
$tableCopy sort -real x
puts [$tableCopy keys]
puts [$tableCopy cget x]; # table access reflects sorted keys
puts [$tableCopy search -sorted -bisect -real x 5.0]
```

Output:

```
3
5 1 2 4 3
3.26 3.44 4.61 5.20 8.25
2
```

Merging Tables

Data from other tables can be merged into the table object with *$tableObj merge*. In order to merge, all the tables must have the same keyname and fieldname. If the merge is valid, the table data is combined, with later entries taking precedence. Additionally, the keys and fields are combined, such that if a key appears in any of the tables, it is in the combined table.

```
$tableObj merge $object ...
```

$object ...	Other table objects to merge into table. Does not destroy the input tables.

Example 65: Merging data from other tables

Code:

```
$tableObj --> tableCopy
table newTable
$newTable set 1 x 5.00 q 6.34
$tableCopy merge $newTable
$tableCopy print
```

Output:

```
key {1 2 3 4 5} x {5.00 4.61 8.25 5.20 3.26} y {7.11 1.81 7.56 6.78 9.92} z {8.67 7.63 3.84
    1.11 4.56} q {6.34 {} {} {} {}}
```

Table Manipulation

The following methods are useful for adding, removing, and rearranging rows and columns in a table.

Overwriting Keys/Fields

The method *define* overwrites the keys and fields of the table, filtering the data or adding keys and fields as necessary. For example, if the keys are defined to be a subset of the current keys, it will filter the data to only include the key subset.

```
$tableObj define keys $keys
$tableObj define fields $fields
```

`$keys`	Unique list of keys.
`$fields`	Unique list of fields.

Adding or Removing Keys/Fields

The method *add* adds keys or fields to a table, appending to the end of the key/field lists. If a key or field already exists it is ignored. The method *remove* removes keys or fields and their corresponding rows and columns from a table. If a key or field does not exist, it is ignored.

```
$tableObj add keys $key ...
$tableObj add fields $field ...
```

```
$tableObj remove keys $key ...
$tableObj remove fields $field ...
```

`$key ...`	Keys to add/remove.
`$field ...`	Fields to add/remove.

Inserting Keys/Fields

The method *insert* inserts keys or fields at a specific row or column index. Input keys or fields must be unique and must not already exist.

```
$tableObj insert keys $index $key ...
$tableObj insert fields $index $field ...
```

$index	Row/column index to insert at.
$key ...	Keys to insert.
$field ...	Fields to insert.

Renaming Keys/Fields

The method *rename* renames keys or fields. Old keys and fields must exist. Duplicates are not allowed in old and new key/field lists.

```
$tableObj rename keys $old $new
$tableObj rename fields $old $new
```

$old	Keys/fields to rename. Must exist.
$new	New keys/fields. Must be same length as $old.

Example 66: Renaming fields

Code:

```
$tableObj --> tableCopy
$tableCopy rename fields [string toupper [$tableCopy fields]]
puts [$tableObj fields]
puts [$tableCopy fields]
```

Output:

```
x y z
X Y Z
```

Moving Keys/Fields

Existing keys and fields can be moved with the method *move*.

```
$tableObj move key $key $index
$tableObj move field $field $index
```

$key	Key to move.
$field	Field to move.
$index	Row/column index to move to.

Swapping Keys/Fields

Existing keys and fields can be swapped with the method *swap*. To swap the a field column with the key column, use the method *mkkey*.

```
$tableObj swap keys $key1 $key2
$tableObj swap fields $field1 $field2
```

$key1 $key2	Keys to swap.
$field1 $field2	Fields to swap.

Example 67: Swapping table rows

Code:

```
$tableObj --> tableCopy
$tableCopy swap keys 1 4
$tableCopy print
```

Output:

```
key {4 2 3 1 5} x {5.20 4.61 8.25 3.44 3.26} y {6.78 1.81 7.56 7.11 9.92} z {1.11 7.63 3.84
    8.67 4.56}
```

Making a Field the Key of a Table

The method *mkkey* makes a field the key of a table, and makes the key a field. If a field is empty for some keys, those keys will be lost. Additionally, if field values repeat, only the last entry for that field value will be included. This method is intended to be used with a field that is full and unique, and if the keyname matches a field name, this command will return an error.

```
$tableObj mkkey $field
```

`$field`	Field to swap with key.

Transposing a Table

The method *transpose* transposes the table, making the keys the fields and the fields the keys.

```
$tableObj transpose
```

Example 68: Transposing a table

Code:

```
$tableObj --> tableCopy
$tableCopy transpose
$tableCopy print
```

Output:

```
key {x y z} 1 {3.44 7.11 8.67} 2 {4.61 1.81 7.63} 3 {8.25 7.56 3.84} 4 {5.20 6.78 1.11} 5
    {3.26 9.92 4.56}
```

File Import/Export

The commands *readFile* and *putsFile* simplify file I/O in Tcl.

```
readFile <$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.

```
putsFile <$option $value ...> <-nonewline> $file $string
```

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

Example 69: File import/export

Code:

```
# Export data to file (creates or overwrites the file)
putsFile example.txt "hello world"
# Import the contents of the file (requires that the file exists)
puts [readFile example.txt]
```

Output:

```
hello world
```

Data Conversion

This package also provides conversion utilities for different datatypes. The main datatype is matrix, or **mat**.

Matrix (mat)

The matrix (**mat**) datatype is a nested Tcl list, where each list element represents a row vector of equal length. This definition is compatible with the matrix data type provided by the ndlist package.

An example of a matrix with headers is shown below.

Example 70: Example data (**mat**):

Code:

```
set mat {{step disp force} {1 0.02 4.5} {2 0.03 4.8} {3 0.07 12.6}}
```

This format can be converted from and to all other formats, as is illustrated in the diagram below, with "**a**" and "**b**" acting as placeholders for all other datatypes.

This way, each new datatype only requires the addition of two new conversion commands: one to **mat** and one from **mat**. Then, you can convert between any datatype using **mat** as the intermediate datatype.

Table (tbl)

The table (**tbl**) datatype is a key-value paired list, with keys representing the table header, and values representing the columns. This definition is compatible with the table data type provided by the taboo package. To convert between **mat** and **tbl**, use the commands *mat2tbl* and *tbl2mat*.

```
mat2tbl $mat
```

```
tbl2mat $tbl
```

$mat	Matrix value.
$tbl	Table value.

Example 71: Example data (**tbl**):

Code:

```
puts [mat2tbl $mat]
```

Output:

```
step {1 2 3} disp {0.02 0.03 0.07} force {4.5 4.8 12.6}
```

Space-Delimited Text (txt)

The space-delimited text (**txt**) datatype is simply space-delimited values, where new lines separate rows. Escaping of spaces and newlines is consistent with Tcl rules for valid lists. To convert between **mat** and **txt**, use the commands *mat2txt* and *txt2mat*.

```
mat2txt $mat
```

```
txt2mat $txt
```

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

Example 72: Example data (**txt**):

Code:

```
puts [mat2txt $mat]
```

Output:

```
step disp force
1 0.02 4.5
2 0.03 4.8
3 0.07 12.6
```

Comma-Separated Values (csv)

The comma-separated values (**csv**) datatype is comma delimited values, where new lines separate rows. Commas and newlines are escaped with quotes, and quotes are escaped with double-quotes. To convert between **mat** and **csv**, use the commands *mat2csv* and *csv2mat*.

```
mat2csv $mat
```

```
csv2mat $csv
```

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

Example 73: Example data (**csv**):

Code:

```
puts [mat2csv $mat]
```

Output:

```
step,disp,force
1,0.02,4.5
2 0.03,4.8
3,0.07,12.6
```

Derived Conversions

Using the **mat** datatype as the intermediate datatype, data can be converted to and from any datatype. As a convenience, shortcuts are provided for conversions that use **mat** as an intermediate data format.

```
tbl2txt $tbl
tbl2csv $tbl
```

```
txt2tbl $txt
txt2csv $txt
```

```
csv2tbl $csv
csv2txt $csv
```

$tbl	Table value.
$txt	Space-delimited values.
$csv	Comma-separated values.

Example 74: Combining data conversions

Code:

```
# Convert from table to csv, using mat as an intermediate datatype.
set tbl {step {1 2 3} disp {0.02 0.03 0.07} force {4.5 4.8 12.6}}
set csv [mat2csv [tbl2mat $tbl]]; # also could use tbl2csv
puts $csv
```

Output:

```
step,disp,force
1,0.02,4.5
2,0.03,4.8
3,0.07,12.6
```

Command Index