N-Dimensional Lists (ndlist)

Version 0.2

Alex Baker

https://github.com/ambaker1/ndlist

September 28, 2023

Abstract

The “ndlist” module provides tools for vector, matrix, and tensor manipulation and processing, where vectors are represented by Tcl lists, and matrices are represented by nested Tcl lists, and higher dimension lists represented by additional levels of nesting. Additionally, this package provides the “ndlist” object variable type, using the object variable framework provided by the vutil package.

N-Dimensional Lists

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. For example, a matrix is a 2D list, or a list of equal length row vectors (1D), which contain arbitrary scalar values, as shown below:

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

Example 1: Defining matrices in Tcl

Code:

```
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}}
```

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".

Initialization

The command *nrepeat* can be used to initialize a valid ND list of any size.

```
nrepeat $shape $value
```

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

Example 2: Create nested ND list with one value

Code:

```
puts [nrepeat {1 2 3} 0]
```

Output:

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

The command *ndlist* initializes an ND list from a value input, expanding if necessary to match maximum dimensions. If the input value is "ragged", as in it has inconsistent dimensions, it will be expanded to match the maximum dimensions, filled in with blanks.

```
ndlist $nd $value
```

`$nd`	Dimensionality of ND list (e.g. 2D for a matrix).
`$value`	List to interpret as an ndlist

Example 3: Expand ragged ND list

Code:

```
puts [ndlist 2D {1 {2 3}}]
```

Output:

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

ND List Access

Portions of an ND list can be accessed with the command *nget*, using index notation as described on pg. 17.

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

`$ndlist`	ND list value.
`$i ...`	Index arguments, using index notation (see pg. 17). The number of index arguments determines the interpreted dimensions.

Example 4: ND list access

Code:

```
set A {{1 2 3} {4 5 6} {7 8 9}}
puts [nget $A 0 :]
puts [nget $A 0* :]; # can "flatten" row
puts [nget $A 0:1 1]
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
2 5
{9 8 7} {6 5 4} {3 2 1}
2 2 2
```

ND List Modification

A ND list can be modified by reference with *nset*, and by value with *nreplace*, using the same index argument syntax as *nget*. If the blank string is used as a replacement value, it will remove values from the ND lists, as long as it is only removing along one dimension. Otherwise, the replacement ND list must agree in dimension to the to the index argument dimensions, or be unity. For example, you can replace a 4x3 portion of a matrix with 4x3, 4x1, 1x3, or 1x1 matrices. If modifying outside of the dimensions of the ND list, the ND list will be expanded to the new dimensions, like in the command *ndlist*.

```
nset $varName $arg1 $arg2 ...  $sublist
```

```
nreplace $ndlist $arg1 $arg2 ...  $sublist
```

`$varName`	Name of ND list to modify.
`$ndlist`	ND list to modify.
`$arg1 $arg2 ...`	Index arguments, using index notation (see pg. 17). The number of index arguments determines the interpreted dimensions.
`$sublist`	Replacement list, or blank to delete values.

Example 5: Swapping rows in a matrix

Code:

```
# ND List Value Modification
set a {{1 2} {3 4} {5 6}}
puts [nreplace $a : 1 ""]; # Delete a column (modify in-place)
nset a {1 0} : [nget $a {0 1} :]; # Swap rows and columns (modify by reference)
puts $a
```

Output:

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

ND 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 agree in dimension or be unity, like in *nset*. Returns an ND list in similar fashion to the Tcl *lmap* command. Additionally, elements can be skipped with *continue*, and the entire loop can be exited with *break*.

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

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

Map Index Access

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

```
i <$axis>
```

```
j
```

```
k
```

`$axis`	Dimension to access mapping index at. Default 0.

Example 6: ND list mapping

Code:

```
set testmat {{1 2 3} {4 5 6} {7 8 9}}
# Checkerboard sign pattern
puts [nmap 2D x $testmat {expr {
    $x*([i]%2 + [j]%2 == 1?-1:1)
}}]
# Simple formatting
puts [nmap 2D x $testmat {format %.2f $x}]
```

Output:

```
{1 -2 3} {-4 5 -6} {7 -8 9}
{1.00 2.00 3.00} {4.00 5.00 6.00} {7.00 8.00 9.00}
```

Basic Math Operations

Basic math operators can be mapped over an ND list with the command *nop*. This method is much faster than *nmap* for simple math operations.

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

`$nd`	Dimensionality of ND list (e.g. 2D for a matrix).
`$ndlist`	ND list to perform element-wise operation over.
`$op`	Math operator (using tcl::mathop namespace).
`$arg...`	Operator arguments (see tcl::mathop documentation).

Additionally, this command is built-in to ND list objects with the "`.=`" operator, as shown below.

Example 7: Element-wise operations

Code:

```
# Using ND list values
puts [nop 1D {1 2 3} -]
puts [nop 1D {1 2 3} + 1]
# Using ND list objects
matrix x {{1 2 3} {4 5 6}}
[$x .= {>= 3}] print
```

Output:

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

ND List Objects

This package provides the "ndlist" type class, using the type system provided by the "vutil" package. The command *tensor* creates a new ND list object variable, which uses the class `::ndlist::ndobj`. The commands *matrix*, *vector*, and *scalar* are shorthand for 2D, 1D, and 0D ND list objects.

`tensor $refName $nd <$value>`

`matrix $refName <$value>`

`vector $refName <$value>`

`scalar $refName <$value>`

`$refName`	Name of reference variable to tie to object.
`$nd`	Dimensionality of ND list (e.g. 2D for a matrix).
`$value`	ND list value.

Example 8: Create ND list object

Code:

```
matrix x {{1 2 3} {4 5 6} {7 8 9}}
puts [$x info]
```

Output:

```
exists 1 ndims 2 shape {3 3} type ndlist value {{1 2 3} {4 5 6} {7 8 9}}
```

Standard Methods

Because the ND list 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.

```
$ndlistObj --> $refName
$ndlistObj <- $object
$ndlistObj = $value
$ndlistObj .= $oper
$ndlistObj := $expr
$ndlistObj ::= $body
$ndlistObj info <$field>
$ndlistObj print <-nonewline> <$channelID>
$ndlistObj destroy
```

`$refName`	Reference name to copy to.
`$object`	ND list object.
`$value`	ND list value, passed through the *ndlist* command.
`$oper`	Math operator and arguments, evaluated with *nop*.
`$expr`	Math expression, evaluated with *nexpr*
`$body`	Tcl script to evaluate with *neval*
`$field`	Field to query (fields "shape" and "ndims" added).
`$channelID`	Open channel to print to.

Example 9: ND list object methods

Code:

```
matrix X {{1 2} {3 4}}
$X ::= {format %0.2f $@.}; # Format values
$X print
$X --> Y; # Copy object
$Y .= {+ 1}; # Perform math operation
$Y print
```

Output:

```
{1.00 2.00} {3.00 4.00}
{2.0 3.0} {4.0 5.0}
```

Index Method

The method "@" allows you to access or modify portions of an ND list, using the same index notation used by *nget*, *nreplace*, and *nset*.

```
$ndlistObj @ $i ...  <$op $input>
```

`$ndlistObj`	ND list object.
`$i ...`	Index arguments, using index notation (see pg. 17). The number of index arguments must match the dimensionality of the ND list object.
`$op $input`	Operator to perform on the sublist. Default simply returns value.
`--> $refName`	Create new object from range with reference name `$refName`.
`<- $object`	Assign value of `$object` to range (must have same dimensionality).
`= $value`	Assign `$value` to range after passing through the *ndlist* command.
`.= $oper`	Modify range in place using *nop*.
`:= $expr`	Modify range in place using *nexpr*.
`::= $body`	Modify range in place using *neval*.

Example 10: ND list object access/manipulation

Code:

```
# Access ND List Objects
matrix X {{1 2} {3 4}}
puts [$X @ : 1]; # get column value
$X @ 1* : --> Y; # create row vector (1D list)
$Y @ end .= {* 2}; # double last element of Y
puts [$Y info]
```

Output:

```
2 4
exists 1 ndims 1 shape 2 type ndlist value {3 8}
```

Object Reference Mapping

Similar to the commands *leval* and *lexpr* in the "vutil" package, the commands *neval* and *nexpr* perform element-wise operations over ND list objects. Both use the command *nmap*, so you have access to the index commands i, j, and k. Additionally, these are built into the "`:=`" and "`::=`" ND list operators.

```
neval $body <$nd $ndlist> <"-->" $refName>
```

```
nexpr $expr <$nd $ndlist> <"-->" $refName>
```

`$body`	Tcl script with list object references.
`$expr`	Tcl expression with list object references.
`$nd`	Dimensionality of ND list (e.g. 2D for a matrix).
`$ndlist`	ND list to iterate over, with **`$@.`** reference.
`$refName`	Reference variable to tie resulting ND list to. Default blank returns value.

Example 11: Element-wise expressions

Code:

```
matrix x {{1 2} {3 4} {5 6}}
matrix y 5.0
puts [nexpr {$@x + $@y}]
```

Output:

```
{6.0 7.0} {8.0 9.0} {10.0 11.0}
```

Example 12: Self-operation, using index access commands

Code:

```
matrix x [nrepeat {2 3} 1]
[$x := {$@. * [i]}] print
```

Output:

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

Dimensionality, Shape and Size

ND list objects store the dimensionality of the list, which can be accessed or changed with the method *ndims*.

```
$ndlistObj ndims <$nd>
```

`$nd`	Dimensionality of ND list (e.g. 2D for a matrix).

The shape of an ND list can be accessed with the command *nshape*, and the total size can be accessed with the command *nsize*. For ND list objects, the methods *shape* and *size* access the shape and size of the value stored in the object.

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

```
nsize $nd $ndlist
$ndlistObj size
```

`$nd`	Dimensionality of ND list (e.g. 2D for a matrix).
`$ndlist`	ND list to get dimensions of.
`$axis`	Axis to get dimension along. Blank for all.

Example 13: Shape and size

Code:

```
set x {{1 2} {3 4} {5 6}}
puts [nshape 2D $x]
puts [nsize 2D $x]
# Convert scalar ND list object to matrix
scalar x 5.0
$x ndims 2
puts [$x shape]
```

Output:

```
3 2
6
1 1
```

Flattening and Reshaping

The command *nflatten* flattens an ND list, and the command *nreshape* flattens and reshapes an ND list to a compatible shape. For ND list objects, the methods *flatten* and *reshape* flatten and reshape the value stored in the object.

```
nflatten $nd $ndlist
$ndlistObj flatten
```

$nd	Dimensionality of ND list (e.g. 2D for a matrix).
$ndlist	ND list to flatten.

```
nreshape $nd $ndlist $shape
$ndlistObj reshape $shape
```

$nd	Dimensionality of ND list (e.g. 2D for a matrix).
$ndlist	ND list to flatten and reshape.
$shape	New shape (list of dimensions).

Example 14: Flatten and reshape ND lists

Code:

```
vector x [nflatten 2D {{1 2 3 4} {5 6 7 8}}]
[$x reshape {2 2 2}] print
```

Output:

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

Transpose or Swap Axes

The command *ntranspose* or the method *transpose* swaps axes of an ndlist. By default, it just transposes the matrix representation of the data, swapping rows and columns. For ND list objects, the method *transpose* transposes the value stored in the object.

```
ntranspose $nd $ndlist <$axis1 $axis2>
$ndlistObj transpose <$axis1 $axis2>
```

`$nd`	Dimensionality of ND list (e.g. 2D for a matrix).
`$ndlist`	ND list to manipulate.
`$axis1`	Axis to swap with axis 2 (default 0)
`$axis2`	Axis to swap with axis 1 (default 1)

Example 15: Transposing a matrix

Code:

```
puts [ntranspose 2D {{1 2} {3 4}}]
```

Output:

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

Example 16: Swapping axes of a tensor

Code:

```
tensor x 3D {{{1 2} {3 4}} {{5 6} {7 8}}}
[$x transpose 0 2] print
```

Output:

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

Combine ND Lists

The command *ninsert* allows you to insert a sublist into an ND list at a specified index and axis. Sublist must agree in dimension at all other axes. For ND list objects, the method *insert* inserts a sublist into the value stored in the object.

```
ninsert $nd $ndlist $index $sublist <$axis>
$ndlistObj insert $index $sublist <$axis>
```

$nd	Dimensionality of ND list (e.g. 2D for a matrix).
$ndlist	ND list to manipulate.
$index	Index to insert at.
$axis	Axis to insert at (default 0).

Example 17: Inserting rows and columns in a matrix

Code:

```
# Insert row
puts [ninsert 2D {{1 2 3} {4 5 6} {7 8 9}} 0 {{A B C}}]
# Insert column
matrix x {1 2 3}
$x insert end {4 5 6} 1
$x print
```

Output:

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

Example 18: Stack tensors

Code:

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

Output:

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

Filling Blanks

Modifying an ND list outside of its dimensions automatically expands the list with blanks to match the new shape. The command *nfill* replaces blank values with a specified filler value. For ND list objects, the method *fill* fills blanks in the value stored in the object.

```
nfill $nd $ndlist $filler
$ndlistObj fill $filler
```

$nd	Dimensionality of ND list (e.g. 2D for a matrix).
$ndlist	ND list to iterate over.
$filler	Filler to replace blanks with.

Example 19: Creating an identity matrix

Code:

```
set I ""
for {set i 0} {$i < 3} {incr i} {
    nset I $i $i 1
}
set I [nfill 2D $I 0]
puts $I
```

Output:

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

ND List Indexing

Index input for all ND list access and modification functions (*nget*, *nreplace*, *nset* and the ND list object method “@”) gets passed through the ND list index parser *::ndlist::ParseIndex*.

```
::ndlist::ParseIndex $input $n
```

`$input`	Index input. Options are shown below:
`:`	All indices
`$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), using *range*.
`$iList`	List of indices (e.g. {0 end-1 5} or 3).
`$i*`	Single index with asterisk, “flattens” the ndlist (e.g. 0* or end-3*).
`$n`	Number of elements in list.

Additionally, index range arguments `$start` and `$stop`, all indices in `$iList`, and single indices `$i` get passed through the *::ndlist::Index2Integer* command, which converts `end`±*integer*, *integer*±*integer* and negative wrap-around indexing (where -1 is equivalent to “end”) into normal integer indices.

```
::ndlist::Index2Integer $index $n
```

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

Range Generator

The index range notation "`$start:$step:$stop`" uses the command *range*, which simply generates a list of integer values.

```
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 20: Integer range generator

Code:

```
puts [range 3]
puts [range 0 2]
puts [range 10 3 -2]
# Alternative for-loop
foreach i [range 5] {
    puts $i
}
```

Output:

```
0 1 2
0 1 2
10 8 6 4
0
1
2
3
4
```

Command Index