Tcl Data Analysis (Tda)

Version 0.1.0

Alex Baker

https://github.com/ambaker1/tda

March 22, 2023

Contents

Introduction to Tda

OpenSees is an open-source scripting-based finite-element analysis software, specializing in earthquake engineering simulation [3]. One of the scripting languages for OpenSees is Tcl, or Tool Command Language, which is a high-level, general purpose procedural language [4]. Tda (pronounced "ta-da!") was developed with the OpenSees user in mind, adding much needed data analysis and manipulation tools. It adds native n-dimensional arrays, a tabular datatype, file import/export and datatype conversion utilities, and data visualization tools to Tcl, which are compartmentalized into separate sub-packages of Tda.

Tda version 0.1.0 was written for OpenSees 3.3.0 and Tcl 8.6.10. Tda was originally developed for OpenSees users, but is general enough for any Tcl application.

Copyright and Disclaimer

1. N-Dimensional List Data Structure

Package:	tda::ndlist
Version:	0.1.0

The “ndlist” module provides tools for list, 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.

This datatype definition is consistent with the definition in the Tcllib math::linearalgebra package, which is the standard Tcl linear algebra library [2].

Vectors (1D)

Tcl provides numerous list manipulation utilities, such as *lindex*, *lset*, *lrepeat*, and more. Since vectors are simply Tcl lists, vectors can be created, accessed, and manipulated with standard Tcl list commands such as *list*, *lindex*, and *lset*.

The ndlist module provides additional vector creation and processing commands, especially for numerical lists.

Range Generator

The command *range* simply generates a range of integer values. 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 1.1: 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
```

Generate Linearly Spaced Vector

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

```
linspace $x1 $x2 $n
```

$x1	Starting value
$x2	End value
$n	Number of points

Example 1.2: Linearly spaced vector generation

Code:

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

Output:

```
0.0 0.25 0.5 0.75 1.0
```

Generate Fixed-Spacing Vector

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

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

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

Example 1.3: Intermediate value vector generation

Code:

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

Output:

```
0 0.25 0.5 0.75 1 0.75 0.5 0.25 0
```

Linear Interpolation

The command *linterp* performs linear 1D interpolation.

```
linterp $xq $xp $yp
```

`$xq`	Vector of x values to query
`$xp`	Vector of x points, strictly increasing
`$yp`	Vector of y points, same length as `$xp`

Example 1.4: Linear interpolation

Code:

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

Output:

```
5
-2.92
```

Logical Indexing

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

```
find <$type> $vector <$op $scalar>
```

$type	Search type. Default -all (returns list of matching indices). Other options are -first and -last, which return the first and last matching indices, or -1 if none are found.
$vector	Boolean vector or vector of values to compare.
$op	Comparison operator. Effectively default "!=".
$scalar	Comparison value. Effectively default 0.

Example 1.5: Logical Indexing

Code:

```
puts [find {0 1 0 1 1 0}]
puts [find -first {0.5 2.3 4.0 2.5 1.6 2.0 1.4 5.6} > 2]
puts [find -last {0.5 2.3 4.0 2.5 1.6 2.0 1.4 5.6} > 2]
```

Output:

```
1 3 4
1
7
```

Dot Product

The dot product of two vectors can be computed with *dot*. This function is based on the math::linearalgebra command *dotproduct*.

```
dot $a $b
```

$a	First vector.
$b	Second vector. Must be same length as **$a**.

Cross Product

The cross product of two vectors of length 3 can be computed with *cross*. This function is based on the math::linearalgebra command *crossproduct*.

```
cross $a $b
```

$a	First vector. Must be length 3.
$b	Second vector. Must be length 3.

Norm and Normalize

The norm of a vector can be computed with *norm*, and a vector can be normalized (norm of 1) with *normalize*. These functions are based on the math::linearalgebra commands *norm* and *unitLengthVector*.

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

```
normalize $a <$p>
```

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

Extreme Values

The commands *max* and *min* compute the maximum and minimum values of a vector.

```
max $vector
```

```
min $vector
```

$vector	Vector (at least length 1) to compute statistic of.

Example 1.6: Extreme values

Code:

```
puts [max {-5 3 4 0}]
puts [min {-5 3 4 0}]
```

Output:

```
4
-5
```

As a convenience, the commands *absmax* and *absmin* compute the absolute maximum and minimum values of a vector.

```
absmax $vector
```

```
absmin $vector
```

$vector	Vector (at least length 1) to compute statistic of.

Example 1.7: Absolute maximum values

Code:

```
puts [absmax {-5 3 4 0}]
puts [absmin {-5 3 4 0}]
```

Output:

```
5
0
```

Sum and Product

The commands *sum* & *product* compute the sum and product of a vector.

```
sum $vector
```

```
product $vector
```

`$vector`	Vector (at least length 1) to compute statistic of.

Example 1.8: Sum and product of matrix columns

Code:

```
puts [sum {-5 3 4 0}]
puts [product {-5 3 4 0}]
```

Output:

```
2
0
```

Average Values

The commands *mean* & *median* calculate the mean and median of of a vector. The command *mean* simply sums the values, and divides the sum by the number of values. The command *median* first sorts the values as numbers, and takes the middle value if the number of values is odd, or the mean of the two middle values if the number of values is even.

```
mean $vector
```

```
median $vector
```

`$vector`	Vector (at least length 1) to compute statistic of.

Example 1.9: Mean and median

Code:

```
puts [mean {-5 3 4 0}]
puts [median {-5 3 4 0}]
```

Output:

```
0.5
1.5
```

Variance

The command *variance* calculates variance, and the command *stdev* calculates standard deviation. By default, they compute sample statistics.

```
variance $vector <$pop>
```

```
stdev $vector <$pop>
```

$vector	Vector (at least length 2) to compute statistic of.
$pop	Whether to compute population variance instead of sample variance. Default false.

Example 1.10: Variance and standard deviation

Code:

```
puts [variance {-5 3 4 0}]
puts [stdev {-5 3 4 0}]
```

Output:

```
16.333333333333332
4.041451884327381
```

Matrices (2D)

Matrices are represented in Tcl by nested lists, where each sublist is a row vector. For example, the following matrices are represented in Tcl 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 1.11: 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}}
```

Transposing

The command *transpose* simply swaps the rows and columns of a matrix. This command is based on the math::linearalgebra command *transpose*.

`transpose $A`

`$A`	Matrix to transpose, nxm.

Returns an mxn matrix.

Example 1.12: Transposing a matrix

Code:

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

Output:

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

Flattening and Reshaping

The command *flatten* flattens a matrix to a 1D vector, while the command *reshape* reshapes a 1D vector into a compatible 2D matrix.

```
flatten $matrix
```

$matrix	Matrix to flatten

```
reshape $vector $n $m
```

$vector	Vector to reshape
$n	Number of rows in new matrix
$m	Number of columns in new matrix

Example 1.13: Flattening and reshaping matrices

Code:

```
puts [flatten {{1 2 3} {4 5 6} {7 8 9}}]
puts [reshape {1 2 3 4 5 6} 3 2]
```

Output:

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

Stacking and Augmenting Matrices

The commands *stack* and *augment* can be used to combined matrices, row or column-wise. Matrices can be combined row-wise or column-wise with the commands *stack* & *augment*.

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

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

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

Example 1.14: Combining matrices

Code:

```
puts [stack {{1 2}} {{3 4}}]
puts [augment {1 2} {3 4}]
```

Output:

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

Matrix Multiplication

The command *matmul* performs matrix multiplication for two matrices. Adapted from *matmul* from the Tcllib math::linearalgebra package, with a few additions. First of all, scalars are considered to be valid matrices, and if more than two matrices are provided, the order of multiplication will be optimized, as described in “Introduction to Algorithms” [1].

```
matmul $A $B <$C $D ...>
```

$A	Left matrix, nxq.
$B	Right matrix, qxm.
$C $D ...	Additional matrices to multiply (optional).

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

Example 1.15: 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:

```
24.0 12.0 72.0 75.0
```

Cartesian Product

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 $list1 $list2 ...
```

$list1 $list2 ...	Lists, or vectors, to take Cartesian product of.

Similarly, the command *cartgrid* returns all combinations of input parameters and lists.

```
cartgrid $dict
cartgrid $keys $list <$keys $list ...>
```

$dict	Dictionary of keys and lists.
$keys	List of parameter names.
$list	Parameter value list.

Example 1.16: Nested parameter study without nested loops

Code:

```
dict set params a {1 2}
dict set params b {3 4}
dict set params c {5 6}
foreach line [cartgrid $params] {
    puts $line
}
```

Output:

```
a 1 b 3 c 5
a 1 b 3 c 6
a 1 b 4 c 5
a 1 b 4 c 6
a 2 b 3 c 5
a 2 b 3 c 6
a 2 b 4 c 5
a 2 b 4 c 6
```

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. 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 "ndlist" module provides commands for creation, query, access, modification, and manipulation of ND lists. All general ND list commands are prefixed with "n", and aliases are provided for matrices and vectors, with prefixes "m" and "v". Additionally, shorthand for row and column operations are denoted by prefixes "r" and "c".

Creation

ND lists can be initialized with *nrepeat*. This is similar to *lrepeat*, except that it generates nested lists. Aliases for matrices (2D) and vectors (1D) are provided with the commands *mrepeat* and *vrepeat*.

```
nrepeat $n $m ... $value
```

```
mrepeat $n $m $value
```

```
vrepeat $n $value
```

`$n $m ...`	Shape of ND list.
`$value`	Value to repeat.

Example 1.17: Create nested ND list with one value

Code:

```
nrepeat 2 2 2 0
```

Output:

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

Shape

The shape (dimensions) of an ND list can be queried with *nshape*. Simply takes the list lengths along index zero, assuming that all other sublists are the same length. Aliases for matrices (2D) and vectors (1D) are provided with the commands *mshape* and *vshape*.

```
nshape $ndtype $ndlist <$dim>
```

```
mshape $matrix <$dim>
```

```
vshape $vector
```

$ndtype	Type of ND list. (e.g. 2D for matrix).
$ndlist	ND list to get shape for.
$dim	Dimension to get (e.g. 0 gets number of rows in a matrix). By default returns list of all dimensions.

Access

Portions of an ndlist can be accessed with the command *nget*. Aliases for matrices (2D) and vectors (1D) are provided with the commands *mget* and *vget*, and aliases for accessing matrix rows and columns (using $i* indexing), are provided with the commands *rget* and *cget*.

```
nget $ndlist $arg1 $arg2 ...
```

```
mget $matrix $i $j
```

```
rget $matrix $i
```

```
cget $matrix $j
```

```
vget $vector $i
```

$ndlist	ND list to access
$arg1 $arg2 ...	Index arguments. The number of index arguments determines the interpreted dimensions.

The index arguments are parsed in accordance with the options shown below. In addition to the options shown below, the parser supports `end`±*integer*, *integer*±*integer* and negative wrap-around indexing (where -1 is equivalent to "end").

:	All indices
$start:$stop	Range of indices (e.g. 0:4).
$start:$step:$stop	Stepped range of indices (e.g. 0:2:-2).
$iList	List of indices (e.g. {0 end-1 5}).
$i*	Single index with asterisk, signals to "flatten" at this dimension (e.g. 0*).

Example 1.18: Significance of asterisk index notation

Code:

```
set A {{1 2 3} {4 5 6} {7 8 9}}
puts [mget $A 0 :]
puts [mget $A 0* :]
```

Output:

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

Modification by Reference

A ND list can be modified by reference with *nset*, 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. Aliases for matrices (2D) and vectors (1D) are provided with the commands *mset* and *vset*, and aliases for modifying matrix rows and columns (using $i* indexing), are provided with the commands *rset* and *cset*.

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

```
mset $varName $i $j $submat
```

```
rset $varName $i $subrow
```

```
cset $varName $j $subcol
```

```
vset $varName $i $subvec
```

$varName	Name of ndlist to modify
$arg1 $arg2 ...	Index arguments. The number of index arguments determines the interpreted dimensions.
$sublist	Compatible ND list to replace at the specified indices, or blank to remove values.

Example 1.19: Swapping rows in a matrix

Code:

```
set a {{1 2} {3 4} {5 6}}
nset a {1 0} : [nget $a {0 1} :]
puts $a
```

Output:

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

Note: if attempting to modify outside of the dimensions of the ND list, the ND list will be expanded and filled with the value in the variable `::qvr::ndlist::filler`. By default, the filler is 0, but this can easily be changed.

Modification by Value

In the same fashion as *nset*, an ND list can be modified by value with *nreplace*, returning a new ND list. Aliases for matrices (2D) and vectors (1D) are provided with the commands *mreplace* and *vreplace*, and aliases for modifying matrix rows and columns (using $i* indexing), are provided with the commands *rreplace* and *creplace*.

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

```
mreplace $matrix $i $j $submat
```

```
rreplace $matrix $i $subrow
```

```
creplace $matrix $j $subcol
```

```
vreplace $vector $i $subvec
```

$ndlist	ND list to modify. Returns new ND list.
$arg1 $arg2 ...	Index arguments. The number of index arguments determines the interpreted dimensions.
$sublist	Compatible ND list to replace at the specified indices, or blank to remove values.

Functional Mapping

A functional map can be applied over an ND list with *nmap*. Note that this differs significantly from the Tcl *lmap* command. Aliases for matrices (2D) and vectors (1D) are provided with the commands *mmap* and *vmap*. Aliases for mapping over matrix rows and columns are provided with the commands *rmap* and *cmap*.

```
nmap $ndtype $command $ndlist $arg1 $arg2 ...
```

```
mmap $command $matrix $arg1 $arg2 ...
```

```
rmap $command $matrix $arg1 $arg2 ...
```

```
cmap $command $matrix $arg1 $arg2 ...
```

```
vmap $command $vector $arg1 $arg2 ...
```

$ndtype	Type of ND list. (e.g. 2D for matrix).
$command	Command prefix to map over ND list.
$ndlist	ND list to map with.
$arg1 $arg2 ...	Additional arguments to append to command call.

Example 1.20: Functional mapping

Code:

```
puts [vmap {format %.2f} {1 2 3}]; # Map a command prefix over a vector
puts [vmap max [transpose {{1 2 3} {4 5 6} {7 8 9}}]]; # Get vector of column maximums
puts [cmap max {{1 2 3} {4 5 6} {7 8 9}}]; # Shorthand way to get column maximums
namespace path ::tcl::mathfunc; # Makes all tcl math functions available as commands.
puts [vmap abs {-1 2 -3}]
```

Output:

```
1.00 2.00 3.00
7 8 9
7 8 9
1 2 3
```

Note: the alias for column mapping actually performs a 1D map on the transpose of the matrix, so if performing multiple column maps, it is more efficient to transpose the matrix once and perform row mappings instead.

Looping and Iteration

The command *nfor* is a general purpose looping and iterating 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*. Aliases for matrices (2D) and vectors (1D) are provided with the commands *mfor* and *vfor*.

```
nfor <$ndtype> $dims $body
nfor $ndtype $varName $ndlist <$varName $ndlist ...> $body
```

```
mfor "$n $m" $body
mfor $varName $matrix <$varName $matrix ...> $body
```

```
vfor $n $body
vfor $varName $vector <$varName $vector ...> $body
```

$ndtype	Type of ND list. (e.g. 2D for matrix).
$dims	List of loop dimensions. Must match length with $ndtype if specified.
$varName	Variable name to iterate with.
$ndlist	ND list to iterate over.
$body	Body to evaluate at every iteration.

Index Access

The iteration indices of *nfor* are accessed with the commands *i*, *j*, & *k*.

```
i <$dim>
```

$dim	Dimension to access mapping index at. Default 0.

The commands *j* and *k* are simply shorthand for *i* with dimensions 1 and 2.

```
j
```

```
k
```

Element-Wise Expressions

The command *nexpr* performs element-wise expressions over multiple ND lists, using *nfor*. Aliases for matrices (2D) and vectors (1D) are provided with the commands *mexpr* and *vexpr*.

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

```
mexpr $varName $matrix <$varName $matrix ...> $expr
```

```
vexpr $varName $vector <$varName $vector ...> $expr
```

$ndtype	Type of ND list. (e.g. 2D for matrix).
$varName	Variable name to iterate with.
$ndlist	ND list to iterate over.
$expr	Tcl expression to evaluate at every loop iteration.

Example 1.21: Various uses of *nexpr*

Code:

```
set testmat {{1 2 3} {4 5 6} {7 8 9}}
# Simple negation
puts [nexpr 2D x $testmat {-$x}]
# Checkerboard
puts [nexpr 2D x $testmat {
    $x*([i]%2 + [j]%2 == 1?-1:1)
}]
# Addition with column vector
puts [nexpr 2D x $testmat y {.1 .2 .3} {$x + $y}]
# Addition with row vector (using tcl::mathfunc::y)
puts [nexpr 2D x $testmat y {{.1 .2 .3}} {$x + $y}]
# Filter a vector using ``continue'' command (note, continue only continues at the lowest
    dimension).
set cutoff 3; # supports local variables in expr.
puts [nexpr 1D x {1 2 3 4 5 6} {$x > $cutoff ? [continue] : $x}]
```

Output:

```
{-1 -2 -3} {-4 -5 -6} {-7 -8 -9}
{1 -2 3} {-4 5 -6} {7 -8 9}
{1.1 2.1 3.1} {4.2 5.2 6.2} {7.3 8.3 9.3}
{1.1 2.2 3.3} {4.1 5.2 6.3} {7.1 8.2 9.3}
1 2 3
```

Element-Wise Operations

If only performing a simple math operation with ND lists, the command *nop* can be used in lieu of *nexpr*. There are three ways to call *nop*, for single argument operations, operations with scalars, and element-wise operations. If performing element-wise operations, ND lists must be compatible in dimension just like in *nset* and *nexpr*. Aliases for matrices (2D) and vectors (1D) are provided with the commands *mop* and *vop*.

```
nop $ndtype $op $ndlist
nop $ndtype $ndlist $op $scalar
nop $ndtype $ndlist1 .$op $ndlist2
```

```
mop $op $matrix
mop $matrix $op $scalar
mop $matrix1 .$op $matrix2
```

```
vop $op $vector
vop $vector $op $scalar
vop $vector1 .$op $vector2
```

$ndtype	Type of ND list. (e.g. 2D for matrix).
$ndlist	ND list to perform element-wise operation over.
$op	Math operator (using tcl::mathop namespace).
$scalar	Scalar to perform operation with.

Example 1.22: Element-wise operations

Code:

```
puts [nop 1D  - {1 2 3}]
puts [nop 1D {1 2 3} + 1]
puts [nop 1D {1 2 3} .+ {3 2 1}]
```

Output:

```
-1 -2 -3
2 3 4
4 4 4
```

2. Tabular Data Structure

Package: tda::tbl

Version: 0.1.0

The “tbl” module implements an object-oriented tabular datatype in Tcl. This datatype is suitable for row-oriented two-dimensional data, and efficiently handles sparse tables.

This is achieved internally by representing the table by five properties: “keyname”, “fieldname”, “keys”, “fields”, and “data”. The property “keyname” describes what the table keys represent, and the property “fieldname” describes what the table fields represent (this is not typically present in raw table formats such as CSV). The property “keys” is an ordered list of all the row names of the table, and the property “fields” is an ordered list of all the field names of the table. The property “data” stores the table values in an unordered nested dictionary, with the first level data keys corresponding to the table keys, and the second level data keys corresponding to the table fields. The conceptual layout of the five properties of a table is illustrated in the figure below.

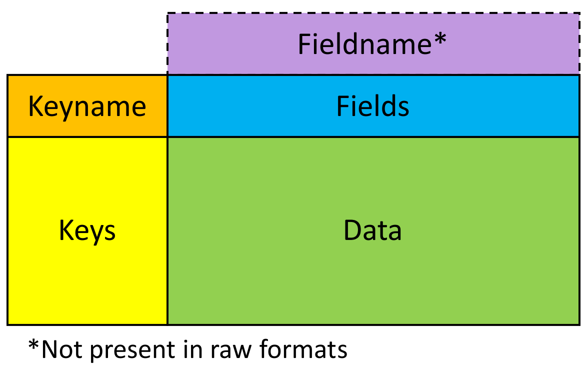


Figure 2.1: The five properties of a table

Creating Table Objects

Table objects are created from the *tdatbl* class using the standard methods *new* or *create*. Once created, table objects act as commands with an ensemble of subcommands, or methods. These objects can be copied with the method *copy* and deleted with the method *destroy*.

```
tdatbl new $arg1 $arg2 ...
tdatbl create $objectName $arg1 $arg2 ...
```

`$objectName`	Explicit name for object.
`$arg1 $arg2 ...`	Arguments to pass to *define* method.

Example 2.1: Creating a table object

Code:

```
set tableObj [table new]
```

Copying Table Objects

The method *copy* copies all the data from a table object to a new object.

```
$tdatblObj copy <$objectName>
```

`$objectName`	Explicit name for object. By default, returns an auto-generated name.

Removing Table Objects

The standard method *destroy* removes a table object from the interpreter.

```
$tdatblObj destroy
```

Table Definition

The method *define* sets the property values of a 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 fields, it will filter the data to only include the key subset. Also, if the data is defined, all existing data will be wiped, and any new keys or fields will be added.

```
$tdatblObj define <$properties> <$option $value ...>
```

$properties	Dictionary of properties. Mutually exclusive with option-value syntax.
$option	Property to set: "keyname", "fieldname", "keys", "fields" or "data".
$value	Value to set property to.

The remaining examples in this documentation will use the table as defined below:

Example 2.2: Example Table

Code:

```
set tableObj [table new]
$tableObj define data {
    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}
}
```

Table Property Query

The method *properties* simply returns a dictionary of the table properties, as defined with *$tdatblObj define*. Additionally, calling the table object without any arguments will return the table properties.

```
$tdatblObj properties
```

Example 2.3: Getting table properties (and trimming a table)

Code:

```
puts [$tableObj properties]; # Automatically generates keys and fields
$tableObj define keys {1 2} fields x; # Trims data
puts [$tableObj properties]
puts [$tableObj]
```

Output:

```
keyname key fieldname field keys {1 2 3 4 5} fields {x y z} data {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}}
keyname key fieldname field keys {1 2} fields x data {1 {x 3.44} 2 {x 4.61}}
keyname key fieldname field keys {1 2} fields x data {1 {x 3.44} 2 {x 4.61}}
```

Get Keyname and Fieldname

The keyname and fieldname properties of a table can be accessed directly with their respective methods.

```
$tdatblObj keyname
```

```
$tdatblObj fieldname
```

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. In addition to just returning the lists of keys and fields, a pattern can be specified in the same style as the Tcl *string match* command.

```
$tdatblObj keys <$pattern>
```

```
$tdatblObj fields <$pattern>
```

$pattern	String matching pattern. Default returns all

Get Table Data (Dictionary Form)

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

```
$tdatblObj data <$key>
```

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

Example 2.4: Getting table data

Code:

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

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
```

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.

```
$tdatblObj values
```

Example 2.5: Getting table values

Code:

```
puts [$tableObj values]
```

Output:

```
{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}
```

Get Table Dimensions

The dimensions of a table, as in the number of keys and fields, can be accessed with the method *shape*. Note that rows and columns with missing data will be counted.

```
$tdatblObj shape <$dim>
```

$dim	Dimension to take size along (default will return the number of rows and columns as a list) 0: Number of rows 1: Number of columns

Alternatively, the number of rows can be queried with *$tdatblObj height* and the number of columns can be queried with *$tdatblObj width*.

```
$tdatblObj height
```

```
$tdatblObj width
```

Example 2.6: Getting table dimensions

Code:

```
puts [$tableObj shape]
puts [$tableObj height]
puts [$tableObj width]
```

Output:

```
5 3
5
3
```

Check Existence of Table Keys/Fields

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

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

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

Find Table Keys/Fields

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 -1.

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

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

Get Table Key/Field

The table key or field corresponding with a row or column index (`end-`*`integer`* format supported) can be queried with the methods *key* and *field*.

```
$tdatblObj key $rid
```

```
$tdatblObj field $cid
```

$rid	Row index.
$cid	Column index.

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 *$tdatblObj set*.

```
$tdatblObj set $key $field $value ...
```

```
$tdatblObj get $key $field
```

$key	Key of row to set/get data in/from.
$field	Field of column to set/get data in/from.
$value	Value to set.

Example 2.7: Setting multiple values

Code:

```
$tableObj set 1 x 2.00 y 5.00 foo bar
puts [$tableObj 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.

```
$tdatblObj rset $key $row
```

```
$tdatblObj rget $key
```

$key	Key of row to set/get.
$row	List of values (or scalar) to set.

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.

```
$tdatblObj cset $field $column
```

```
$tdatblObj cget $field
```

$field	Field of column to set/get.
$column	List of values (or scalar) to set.

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. Note that *$tdatblObj mget* with no arguments is identical to *$tdatblObj values*.

```
$tdatblObj mset <$keys $fields> $matrix
```

```
$tdatblObj mget <$keys $fields>
```

$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.

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.

`$tdatblObj with $body`

`$body`	Code to execute.

Example 2.8: Iterating over a table, accessing and modifying field values

Code:

```
set a 20.0
$tableObj add fields q
$tableObj with {
    puts [list $key $x]; # access key and field value
    set q [expr {$x*2 + $a}]; # modify field value
}
puts [$tableObj 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 *$tdatblObj 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 ($), the "at" symbol (@) is used by *$tdatblObj 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.

`$tdatblObj 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.

`$tdatblObj fedit $field $fieldExpr`

`$field`	Field to set.
`$fieldExpr`	Field expression.

Example 2.9: Using field expressions

Code:

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

```
$tdatblObj query $fieldExpr
```

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

Example 2.10: 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.

```
$tdatblObj filter $fieldExpr
```

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

Example 2.11: Filtering table to only include keys that match criteria

Code:

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

Output:

```
1 3 5
```

Searching a Table

Besides searching for specific field expression criteria with *$tdatblObj 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.

```
$tdatblObj search <$option1 $option2 ...> <$field> $value
```

$option1 $option2 ...	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.

```
$tdatblObj sort <$option1 $option2 ...> <$field1 $field2 ...>
```

$option1 $option2 ...	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
$field1 $field2 ...	Fields to sort by (in order of sorting). If blank, sorts by keys.

Example 2.12: Searching and sorting

Code:

```
puts [$tableObj search -real x 8.25]; # returns first matching key
$tableObj sort -real x
puts [$tableObj keys]
puts [$tableObj cget x]; # table access reflects sorted keys
puts [$tableObj 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 *$tdatblObj 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.

```
$tdatblObj merge $arg1 $arg2 ...
```

`$arg1 $arg2 ...`	Other table objects to merge into table. Does not destroy the input tables.

Example 2.13: Merging data from other tables

Code:

```
set newTable [table new]
$newTable set 1 x 5.00 q 6.34
$tableObj merge $newTable
$newTable destroy; # clean up
puts [$tableObj properties]
```

content...

Output:

```
keyname key fieldname field keys {1 2 3 4 5} fields {x y z q} data {1 {x 5.00 y 7.11 z 8.67
    q 6.34} 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}}
```

Table Manipulation

The following methods are useful for adding, removing, and rearranging rows and columns in a table. With the exception of *$tdatblObj remove*, which removes corresponding data, and *$tdatblObj mkkey*, which may cause data loss, these methods do not add or remove data, they only modify the key and field lists.

Adding 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.

```
$tdatblObj add keys $arg1 $arg1 ...
$tableObj add fields $field1 $field2 ...
```

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

Removing Keys/Fields

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.

```
$tdatblObj remove keys $key1 $key2 ...
$tableObj remove fields $field1 $field2 ...
```

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

Cleaning a Table

Keys and fields with no data are removed with the method *clean*.

```
$tdatblObj clean
```

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.

```
$tdatblObj insert keys $rid $key1 $key2 ...
$tableObj insert fields $cid $field1 $field2 ...
```

$rid	Row index to insert keys at.
$key1 $key2 ...	Keys to remove.
$cid	Column index to insert fields at.
$field1 $field2 ...	Fields to remove.

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.

```
$tdatblObj rename keys $oldKeys $newKeys
$tableObj rename fields $oldFields $newFields
```

$oldKeys	Keys to rename. Must exist.
$newKeys	New key names. Must be same length as $oldKeys.
$oldFields	Fields to rename. Must exist.
$newFields	New field names. Must be same length as $oldFields.

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.

```
$tdatblObj mkkey $field
```

$field	Field to swap with key.

Swapping Rows/Columns

Existing rows and columns can be swapped with the methods *rswap* and *cswap*.

```
$tdatblObj rswap $key1 $key2
```

```
$tdatblObj cswap $field1 $field2
```

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

Moving Rows/Columns

Existing rows and columns can be moved with the methods *rmove* and *cmove*.

```
$tdatblObj rmove $key $rid
```

```
$tdatblObj cmove $field $cid
```

$key	Key of row to move.
$rid	Row index to move to.
$field	Field of row to move.
$cid	Column index to move to.

Transposing a Table

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

```
$tdatblObj transpose
```

Example 2.14: Transposing a table

Code:

```
$tableObj transpose
puts [$tableObj properties]
```

Output:

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

3. Datatype Conversion and File Utilities

Package:	tda::io
Version:	0.1.0

The “io” module provides data import, export and datatype conversion. Four datatypes are supported by the “io” module: space-delimited values (txt), comma-separated values (csv), nested Tcl lists, or matrices (mat), and Tda tables (tbl).

File Utilities

The commands *readFile*, *writeFile*, and *appendFile* simplify reading and writing of files in Tcl. Syntax is inspired from similar commands in the Tcllib fileutil package.

readFile <$option $value ...> <-newline> $filename

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

writeFile <$option $value ...> <-nonewline> $filename $data

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

appendFile <$option $value ...> <-nonewline> $filename $data

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

Example 3.1: File import/export

Code:

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

Output:

```
hello world
goodbye moon
```

Data Conversion

The “io” module provides conversion utilities for different datatypes. The intermediate format for Tda data conversion is matrix, or **mat**.

Matrix (mat)

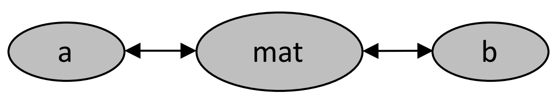
The matrix (**mat**) datatype is a nested Tcl list, where each list element represents a row vector of equal length. The “io” module is based around the **mat** datatype. An example of a matrix with headers is shown below.

Example 3.2: Example Data (**mat**):

Code:

```
{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” & “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**.

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. This is the datatype outputted by OpenSees recorders with the -file option. An example of the same data from the matrix example in **txt** format is shown below.

Example 3.3: Example Data (**txt**):

Code:

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

To convert between **mat** & **txt**, use the commands *mat2txt* & *txt2mat*.

```
mat2txt $mat <$includeHeaders> <$includeRownames>
```

`$mat`	Tcl matrix
`$includeHeaders`	Boolean, whether to include first row of the matrix. Default true
`$includeRownames`	Boolean, whether to include first column of the matrix. Default true

```
txt2mat $txt <$includeHeaders> <$includeRownames>
```

`$txt`	Space & newline-delimited table
`$includeHeaders`	Boolean, whether to include first row of the text data. Default true
`$includeRownames`	Boolean, whether to include first column of the text data. Default true

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. This datatype is commonly used in post-processing and plotting programs, such as MS Excel. An example of the same data from the matrix example in **csv** format is shown below.

Example 3.4: Example Data (csv):

Code:

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

To convert between **mat** & **csv**, use the commands *mat2csv* & *csv2mat*.

```
mat2csv $mat <$includeHeaders> <$includeRownames>
```

`$mat`	Matrix
`$includeHeaders`	Boolean, whether to include first row of the matrix. Default true
`$includeRownames`	Boolean, whether to include first column of the matrix. Default true

```
csv2mat $csv <$includeHeaders> <$includeRownames>
```

`$csv`	Comma-separated values (with escaped commas, newlines, and quotes)
`$includeHeaders`	Boolean, whether to include headers from CSV data. Default true
`$includeRownames`	Boolean, whether the CSV data has rownames. Default true

Table (tbl)

The table (**tbl**) datatype represents tabular data with row and column names, and are created and manipulated with the table module. If headers or row names are not included when converting to tabular data, default keys and fields will be generated, with keys starting at 0 and fields starting at "A". To convert between "mat" & "tbl", use the commands *mat2tbl* & *tbl2mat*.

`mat2tbl $mat <$includeHeaders> <$includeRownames>`

`$mat`	Matrix
`$includeHeaders`	Boolean, whether to include first row of the matrix. Default true
`$includeRownames`	Boolean, whether to include first column of the matrix. Default true

`tbl2mat $tableObj <$includeHeaders> <$includeRownames>`

`$tableObj`	Table object name
`$includeHeaders`	Boolean, whether to include table fields as headers. Default true
`$includeRownames`	Boolean, whether to include table keys as the first column. Default true

Conversion Shortcuts

Using the **mat** datatype as the intermediate datatype, data can be converted to and from any datatype, as is shown in the example below.

Example 3.5: Example Code (using **mat** as intermediate):

Code:

```
set txt {step disp force
1 0.02 4.5
2 0.03 4.8
3 0.07 12.6}
set mat [txt2mat $txt]
set csv [mat2csv $mat]
puts $csv
```

Output:

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

For data conversions that use matrix as an intermediate format, shortcut commands are provided, illustrated in the table below. The optional arguments `$includeHeaders` & `$includeRownames` are the same for the shortcut conversion commands.

	txt	**csv**	**tbl**
txt		txt2csv	txt2tbl
csv	csv2txt		csv2tbl
tbl	tbl2txt	tbl2csv	

One application of a shortcut conversion is bulk conversion of all recorder output files to **csv**.

Example 3.6: Example Code (convert .out files to .csv):

Code:

```
foreach filename [glob *.out] {
    writeFile [file rootname $filename].csv [txt2csv [readFile $filename]]
}
```

Data Import and Export Shortcuts

In addition to the fundamental file utilities and data conversion commands, the “data” module also provides some shortcut commands for common data import and export workflows.

Matrix Import and Export

The commands *readMatrix* and *writeMatrix* read/write a matrix from/to a file, converting from/to **csv** if the extension is .csv, and converting from/to **txt** otherwise. Except for the input argument **`$matrix`**, the input arguments are identical to *readFile* and *writeFile*.

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

```
writeMatrix <$option $value ...> <-nonewline> $filename $matrix
```

`$matrix`	Matrix to convert and write to file.

Table Import and Export

The commands *readTable* and *writeTable* read/write a table from/to a file, converting from/to **csv** if the extension is .csv, and converting from/to **txt** otherwise. Except for the input argument **`$table`**, the input arguments are identical to *readFile* and *writeFile*.

```
readTable <$option $value ...> <-newline> $filename
```

```
writeTable <$option $value ...> <-nonewline> $filename $table
```

`$table`	Table to convert and write to file.

4. Data Visualization

Package: tda::vis

Version: 0.1.0

The "vis" module provides utilities for viewing data in Tcl. It utilizes the "wob" package for managing Tk widgets, so in order to interact with the widgets, one must enter the Tcl/Tk event loop. The *mainLoop* command in the "wob" package is an easy way to accomplish this.

View tabular and matrix data

Tda tables and matrices can be interactively explored with the commands *viewTable* and *viewMatrix*.

```
viewTable $tblObj
```

```
viewMatrix $matrix
```

$tblObj	Object name of Tda table object.
$matrix	Matrix to view.

Example 4.1: Viewing tabular and matrix data

Code:

```
set tableObj [tdatbl new]
$tableObj define data {
    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}
}
viewTable $tableObj
viewMatrix [$tableObj values]
wob::mainLoop
```

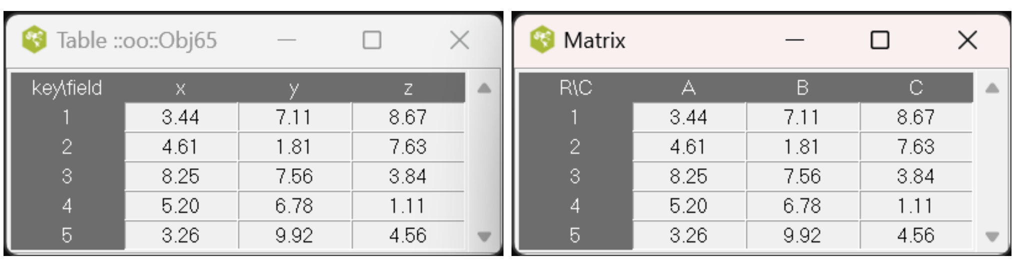


Figure 4.1: Interactive table and matrix viewer

Plot XY data

XY data can be explored graphically with the command *plotXY*. Use the arrow keys or slider to move through the data, and use the scroll wheel on the mouse to switch between data series. This widget was inspired from "plotpoints" on the Tcl wiki: https://wiki.tcl-lang.org/page/A+little+function+plotter.

```
figure $XY
figure $X $Y1 $Y2 ...
```

`$XY`	Two-column matrix, first column X, second column Y1, third column Y2, etc.
`$X $Y1 $Y2 ...`	X and Y vectors of the same length, mutually exclusive with `$XY`.

Example 4.2: Creating a figure object

Code:

```
namespace path ::tcl::mathfunc
set x [linsteps 0.01 -10 10]
set y [vmap sin $x]
plotXY $x $y
wob::mainLoop
```

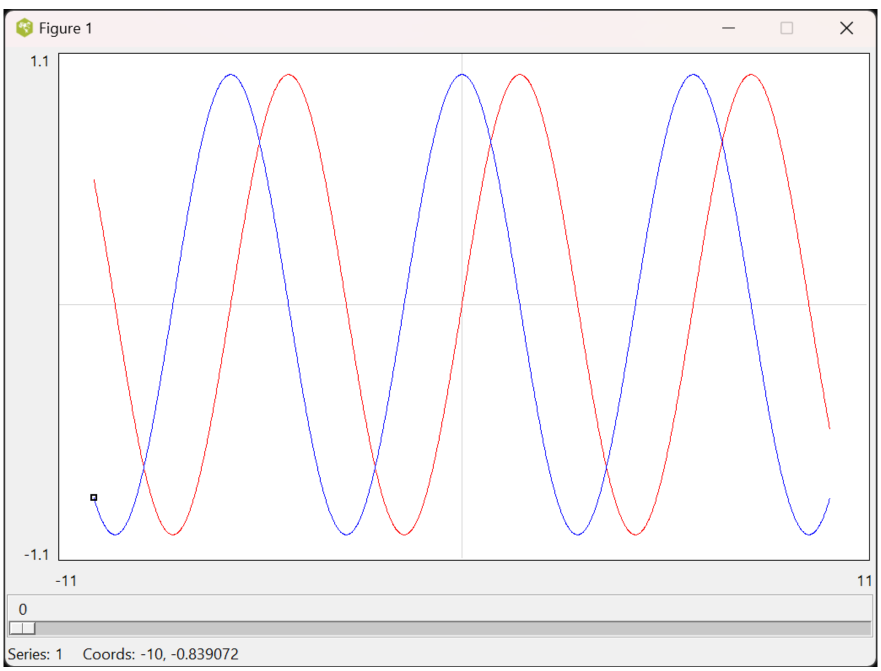


Figure 4.2: Example XY plot

Bibliography

[1] T.H. Cormen et al. *Introduction To Algorithms*. Ed. by T.H. Cormen, MIT Press, and McGraw-Hill Publishing Company. Introduction to Algorithms. MIT Press, 2001. ISBN: 978-0-262-03293-3.

[2] Arjen Markus, Ed Hume, and Michael Buadin. *Tcl Math Library: Linear Algebra*. 2008.

[3] Frank McKenna, Michael H Scott, and Gregory L Fenves. "Nonlinear Finite-Element Analysis Software Architecture Using Object Composition". In: *Journal of Computing in Civil Engineering* 24.1 (2010), pp. 95–107. ISSN: 0887-3801. DOI: 10.1061/(ASCE)CP.1943-5487.0000002.

[4] John K Ousterhout. *Tcl: An Embeddable Command Language*. University of California, Berkeley, Computer Science Division, 1989.

Command Index