March 1998

Mx a package for rapid mathematical prototyping and algorithm development with application to speech and speaker recognition

Sarel van Vuuren

Mx

A Package for Rapid Mathematical Prototyping and Algorithm Development with Application to Speech and Speaker Recognition

Sarel van Vuuren

March 6, 1998

Technical Report

Anthropic Speech Processing Group
Department of Electrical and Computer Engineering
&
Center for Spoken Language Understanging
Department of Computer Science and Engineering

Oregon Graduate Institute of Science and Technology
P.O. Box 91000, Portland, Oregon 97291-1000

Contents

List of Figures

List of Tables

Abstract

We describe a software package called Mx that supports rapid mathematical prototyping and algorithmic development. Benefits of the package include a small memory footprint and high execution speed.

A typical modular approach to research systems, such as for speech and speaker recognition, is a set of precompiled routines, combined in a script and operating on files. Using the extendible script language Tcl, Mx allows a somewhat different approach. Basic functionality is contained in C-code libraries written to be highly portable and fully accessible in the Tcl scripting environment. With Mx, data can stay in memory, precompiled routines are dynamically loaded only once and control flow can be scripted in great detail. In particular, Mx provides extensive scripting capability for the manipulation of matrices and vectors. These data objects are treated as conventional C data structures that are represented at the script level as simple string identifiers.

A unique feature of Mx is that the user can control memory usage using the syntax of the script language. The result is that not only can an algorithm be scripted conveniently, but memory usage can also be tailored for algorithmic efficiency. We provide an overview of Mx, explain the unique syntax, provide examples of usage, and include a detailed summary of the different commands.

Preface

Mx was borne out of the need for a fast and easy to use mathematical package. This package was to have a scripting interface and allow the manipulation of huge amounts of data as is often found in speech and speaker recognition tasks. I wrote this package in the summer of 1995 and has since used it extensively in my research. For example it formed the backbone of the software used in Oregon Graduate Institute's participation in the 1997 NIST Speaker Recognition Evaluation. We have also used it in such forays as investigating novel temporal features for speech recognition.

Over this period Mx has matured and grown until now it more or less satisfies the initial design requirements. Along the way many people have commented on its features, implementation and use. In particular I would like to thank Jacques de Villiers for his supporting advice during the initial design and invaluable support to allow Mx to become part of a much bigger package called CSLUsh; Johan Schalkwyk for using Mx from early on and helping establish it as a mathematics environment for CSLUsh; and Pieter Vermeulen for his encouraging use of Mx. I would also like to thank Ron Cole for gratuitously allowing the bundling of Mx with CSLUsh and Mark Fanty for helping with a Windows NT port of the package. Lastly, my thanks to my advisor Hynek Hermansky who was instrumental in establishing a productive environment for the Mx effort.

This paper consists of four parts. Part I presents Mx and explains how to use it. Part II provides a concise reference to the Mx language. Part III describes the commands in Mx in detail. Finally the Appendix explains how to obtain Mx, provides examples of Mx use and describes the algorithms that were used for the commands.

August 1997, Portland, Oregon — Sarel van Vuuren

Part I

Overview

Chapter 1

Introduction

1.1 Introduction

This document is about a software package called Mx. This package allows rapid mathematical prototyping and algorithm development from within the Tcl shell [1]. It provides a programming system with extensive scripting capability for matrix and vector mathematics. All matrices are abstracted memory objects that are referenced through a unique text-string *handle*.

The Mx package is particularly useful for situations where large matrices need to be manipulated relatively quickly and where rapid prototyping or algorithm development is needed. Examples of such situations include feature processing and modeling of speech [2, 3, 4] at both the research and application development stages. Mx allows fast and easy assimilation, partitioning, transformation and inspection of these matrices. In addition to the usual mathematics operations (real, complex, element-wise and ranging) Mx allows operations on lists such as

```
set x [mx join row [list $a $b $c]].
```

One of the powerful features of Mx is that it allows the user to explicitly control memory usage. Advantages of this include considerable execution time speed-ups and a generally small memory footprint. As an example of such functionality, consider the equation

$$r = \mathrm{cov}(y)\mathrm{cov}(x)^{-1},$$

where x and y are data matrices, and "cov" is the covariance. Furthermore, suppose that this equation will be called many times in a loop so that considerable speed-up may be gained by exploiting the memory usage of temporary variables. The following code fragment details how this may be done using Mx.

```
mx cov $x sx
mx cov $y sy
mx cholinv $sx si
set r(i) [mx prod $sy $si]
```

As depicted in Figure 1.1 the suffix position for the temporary variables `sx`,

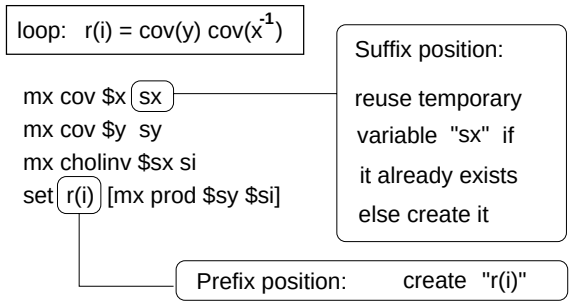

Figure 1.1: Example of controllable memory usage.

`sy`, and `si` indicates to Mx to allocate memory for them only at the first pass of the loop and on subsequent passes to simply overwrite their contents. A prefix position on the other hand indicates to Mx not to reuse memory – as for the variable `r(i)` in the previous example. This implementation allows execution speed to approach that of dedicated C-code. To change the previous code fragment to ignore the first 2 rows of `sy` in the product it suffices to specify a subrange

```
set r(i) [mx prod $sy.(:2,:) $si]
```

Table 1.1 highlights some of the main features of Mx.

- Support for variables, lists and elements in a scripting environment.
- Ability to manipulate large amounts of numeric data.
- Functions to assimilate, partition, transform and inspect numeric data.
- Support for real and complex mathematical operations.
- Support for matrix, vector and scalar arguments.
- Controllable memory usage.
- Error messages and command stack trace.
- Elementary exception handling.
- Remote command execution.

Table 1.1: Mx's features.

1.2 Background

Mx was developed to address the need for a system that would allow rapid mathematical prototyping and algorithm development for speech and speaker recognition systems. To this end it was designed to aid both research and application development.

Existing tools for speech recognition often enforce a methodology where iteratively data is read from disk, some transformation applied in memory and the result written back to disk. This methodology is the result of the basic algorithmic building blocks in such tools requiring input and output to be from disk. However, a significant overhead results from this shuffling of data back and forth from memory to disk. A better strategy might be to design basic algorithmic building blocks that pass data to one another directly via memory using *handles* with access to these building blocks and handles made available within a scripting environment. As far as we are aware this strategy has been available only in generally non-speech related programming environments. Mx is an attempt at making this strategy available to the speech community. To realize this strategy two design considerations were paramount.

One design consideration was that data should be kept in memory as long as possible and passed through an algorithm by means of a reference to the data object (using a *handle* to the object). The other design consideration was that Mx should allow memory usage to be controlled by the user. This is motivated in that algorithms often spend a lot of time managing memory (eg allocating, reallocating or freeing memory) and that a means to shorten time spent thus would allow for enhanced performance. Table 1.2 highlights some of the main benefits gained from using Mx.

Short development cycle using Mx's scripting environment for rapid mathematical prototyping and algorithm development.
High performance by keeping objects in memory and reusing memory on demand.
Modularity and consistency by allowing low-level code sharing in a scripting environment.
Portability by extending Tcl with a highly portable C-code library.
Extendibility by providing a simple developer's API to extend Mx.

Table 1.2: Benefits gained using Mx.

1.3 Software architecture

Mx acts as a package extension of the Tcl script language [1]. The extendible scripting language Tcl is used to access Mx functionality within an efficient scripting environment. Currently Mx is bundled with the *Center for Spoken Language Understanding* programming shell [5] called CSLUsh [6] which is a group of similar package extensions designed specifically as speech recognition tools. This bundling allows CSLUsh to make seamlessly use of Mx and *vice versa*. Individually, each package extends the usability of the other packages by providing specific capabilities such as, for example, distributed computing and object management (using the Rtcl package). In particular, the Rtcl package allows Mx commands to be executed remotely [5]. As with Tcl, Mx also provides error messages, command stack tracing and exception handling. All packages can be dynamically loaded as needed.

The software architecture of Mx consists of two layers, similar to the software architecture of CSLUsh. The bottom layer is a set of efficient C libraries called CSLU-C [7] containing functions which support basic algorithmic operations and associated utilities. The top layer CSLUsh provides a wrapper for the bottom layer CSLU-C libraries, making them accessible within Tcl.

Figure 1.2 shows the relation of Mx to Tcl, CSLU-C and CSLUsh. This document discusses the functional aspects relating to the shaded region in the Figure.

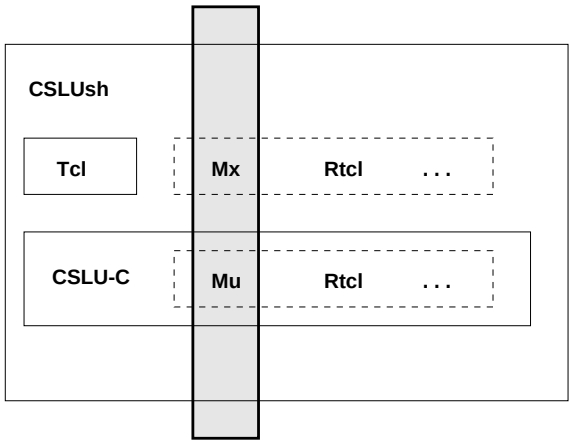

Figure 1.2: The software architecture of Mx.

At the bottom level Mx consists of a stand-alone C-code numerical library of functions. This library will be referred to as Mu (short for "mu-pack"). These functions implement algorithms such as an FFT or matrix addition. This library does not contain any dependencies on Tcl. As such the Mu library is in principle exchangeable and extendible with other numerical libraries.

At the top level Mx consists of a C-code library that provides an interface between the numerical library and the Tcl scripting environment. This top level effectively wraps the numerical routines so that they are accessible within the Tcl scripting environment. This C-code library links with the Tcl library, the latter which contains C functions which a Tcl extended C program may use. It also links with some of the CSLU-C and CSLUsh libraries to allow Mx to be used within the Center for Spoken Language Understanding programming shell CSLUsh.

1.4 Versions

Table 1.3 lists information about the Mx distribution. Mx has been ported to a number of UNIX platforms as well as to Windows NT. This document corresponds to Mx version 2.0 or less.

Programming language:	Tcl script
Implementation language:	C
Syntax:	Pre-fix, command driven
Applicability:	Rapid mathematical prototyping
Author:	Sarel van Vuuren
License:	© Copyright 1996,1997,1998 Sarel van Vuuren General CSLU software license where applicable.
Availability:	http://www.cse.ogi.edu/CSLU/toolkit/
Platforms:	SPARC/Solaris, i86pc/Solaris, Alpha/Digital UNIX, Windows NT, Windows 95.

Table 1.3: The Mx package.

The Mx package can of course not be guaranteed bug free. However, all the commands have been tested and checked extensively. Please email comments, suggestions and bugs reports to the author.

1.5 Prerequisites

Because Mx extends Tcl, a good working knowledge of Tcl is recommended [1]. This document will assume a basic Tcl programming knowledge. In particular it is important to understand how quoting works in Tcl.

1.6 Notation

For the examples to follow, note that in Tcl syntax, a backslash \ denotes a line-break and a pound symbol # a comment string. A percentage symbol % at the start of a line will sometimes be used to denote the command prompt. This will differentiate command input from results, the latter which will not have the leading prompt. We will generally write computer input and output using a `typewriter` font. However, to aid clarity we will sometimes write *arguments* slanted and **keywords** in bold.

If an argument or group of arguments are enclosed in question marks is means they are optional. We'll often name the input(s) using the first letters of the alphabet and the output(s) using the last.

Chapter 2

Using Mx

This Chapter gives an overview on using Mx. It explains how to start and end an Mx session, provides a brief description of the different commands available in Mx and gives simple examples of how to use some of these commands. This Chapter goes on to provide suggestions and examples showing how to exploit Mx's functionality by explicit memory management. In that the user should control (enforce) when and where to create new objects, reuse them or destroy them, in similar way as with a language such as C, it is important that the user understands this aspect of Mx's functionality.

2.1 Matrices

Matrices in Mx are two-dimensional. A vector is viewed as a two-dimensional matrix with one dimension hard-wired to unity. In the examples

```
[1 1 1]   is 1 x 3

[1
 1
 1]       is 3 x 1

[1 1 1
 1 1 1
 1 1 1]   is 3 x 3
```

all the matrices should be thought of as two-dimensional objects.

Matrices are represented contiguously in memory one row after another. Therefore, bookkeeping considerations dictate less computational overhead for row vectors than column vectors and use of row vectors is recommended where appropriate.

Computations in Mx are done in floating point[1]. The global Tcl variable *tcl_precision* determines the number of significant digits that are retained when floating values are converted to strings (except that trailing zeroes are omitted). If *tcl_precision* is unset then 6 digits of precision are used. Internal computations however, are always done in single precision. To set the precision to 3 significant digits type

```
set tcl_precision 3
```

2.2 Getting Started

To use Mx it is necessary to first load the Mx package within a Tcl session by typing the command

```
package require Mx
```

This loads the Mx dynamic library. Mx has a command syntax similar to Tcl. All Mx commands take the following form

`mx` *command args*

A typical Mx argument is a *handle* (a text-string) that refers to a matrix object.

Helpful output can be obtained by typing a nonsense input at the prompt. Here is an example

```
% mx
wrong # args: should be "mx option arg ?arg ...?"
% mx ?
bad option "?": should be fwrite, zeromean, sign, dim, sin ...
```

In addition to the specific Mx commands, some generic commands for manipulating matrix objects are available by loading an auxiliary package called Rtcl (an acronymn for *Remote* Tcl [5]). The Rtcl package can be loaded with the Mx package. The version number of the package is returned as a result.

```
% package require Mx
1.0
% package require Rtcl
1.0
```

Typing the Tcl command

```
exit
```

ends a session.

[1] For historical reasons, mainly storage efficiency, only floating point operations have been supported. This may change in the future.

2.3 Commands and arguments

Commands in Mx are always preceded by the `mx` keyword. This tells the interpreter in the CSLUsh shell that the Mx library applies and an Mx command is to follow. Next the actual command, arguments and any options are specified. Therefore a command takes the form

`mx` *command* ?*subcommand*? *input* ?*input*? ... ?*output*? ?*options*?

where, if an argument or group of arguments are enclosed in question marks it means they are optional.

The commands in Mx can be grouped into several categories. There are commands for manipulating matrices such as selecting a range (`cut`) and joining matrices (`join`). There are commands for writing and reading matrices to and from a variety of channels. There are commands for such basic mathematics as adding (`add`) and taking the product of two matrices (`prod`). There are commands for decompositions such as the Cholesky decomposition `chol` and transformations such as the Fast Fourier Transform (`fft`) and elementary statistical operations such as computing a covariance matrix (`cov`). Table 2.1 lists the Mx commands in terms of their functionality. For a detailed description of each command please refer to Part III.

Some commands can be applied to either rows or columns. These commands take a subcommand (`row` or `col`) specifying row or column-wise operation. Depending on the command the input and output arguments may be real or complex. Generally, real inputs result in a real output value, complex inputs result in a complex output value.

2.4 Using variables

In Mx matrix objects are named using a unique text-string *handle* of the form `arrayF:`n, where n is a unique number. This naming is handled entirely by Mx.

For example, to create two matrices, both with two rows and two columns with ones for the elements

```
% mx ones 2 2
arrayF:0
% mx ones 2 2
arrayF:1
```

Now consider the element-wise addition of the two matrices `arrayF:0` and `arrayF:1`, with the result being a new matrix `arrayF:2`. (Actually we are adding the *contents* of the matrix named `arrayF:0` to the *contents* of the matrix named `arrayF:1` with the result being the *contents* of the matrix named `arrayF:2`.)

Manipulation	', (), copy, cut, join, size, dim, ones, zeros, eye, random, diag, zero, one, zerodiag, onediag, scale, tr, unwrap, flipud, fliplr, load, unload
Writing and reading	set, puts, value, fprintf, fscanf, fwrite, fread
Basic mathematics	trace, prod, mul, div, rem, add, subtr, sqr, abs, angle, real, imag, conj, c2p, p2c (Polar coordinates) pmul, pdiv, padd, psubtr, psqr pabs, pangle, preal, pimag (Numeric) ceil, floor, sign (Trigonometric) cos, sin, tan, acos, asin, atan, cosh, sinh, tanh (Exponential) sqr, sqrt, exp, log, log10
Decompositions and transformations	chol, cholinv, cholsolve, fft, ifft, norm, normalize, qsort
Elementary statistics	find, cov, corr, mean, min, max, sum, std, zeromean

Table 2.1: Mx commands grouped by functionality.

```
% mx add arrayF:0 arrayF:1
arrayF:2
```

And to add this result to itself

```
% mx add arrayF:2 arrayF:2
arrayF:3
```

The `add` command always takes as input the names of two matrices and returns the name of the matrix containing the result. Of course using the rather bulky text-string handles to specify each matrix object would be tedious. An elegant solution is to put each handle into a Tcl variable.

```
% set a arrayF:0
arrayF:0
% set b arrayF:1
arrayF:1
```

Consider again the element-wise addition of the two matrices. Now suppose the handles for these matrices are the contents of two Tcl variables `a` and `b` and that the handle to the result should be stored in the Tcl variable[2] `z`

```
% set z [mx add $a $b]
arrayF:2
```

The dollar sign above (eg. `$a`) is part of the Tcl syntax and tells the interpreter to substitute `$a` with the contents of the variable named `$a`. The square brackets are also part of the Tcl syntax and tells the interpreter to first evaluate the command inside the brackets and then to set the contents of the variable `z` equal to the result – in this case `arrayF:2`. Note that any previous contents of `z` will be overwritten. In this example Mx still expects exactly the same inputs and returns the same output as before as can be seen by looking at the contents of the variables using Tcl's `puts` command

```
% puts $a
arrayF:0
% puts $b
arrayF:1
% puts $z
arrayF:2
```

Because of the convenience of Tcl variables we will almost always use them in this way to hold the handles to Mx matrix objects. Note that this leads to a powerful syntax. For example instead of using simple variable names as in the previous example in some instances we may desire to use associative variables, their contents still being handles to the matrix objects. For example

[2] This is a convenient notation that we'll often use. We name the input(s) using the first letters of the alphabet and the output(s) using the last.

```
set a(red) [mx add $a(green) $a(blue)]
```

2.5 Specifying matrix input

Mx accepts numeric input and can return numeric output. A matrix in Mx is a double list, with the inner lists corresponding to rows of the matrix. Note that the double list specification is only necessary to prevent ambiguity. For example to create a matrix of one element with contents `3.4`

```
mx set 3.4 x
```

suffices. But to create a matrix

```
[  2    3    0.3  0.1
  -4    0.5  4    2
   0.1 -0.2  6    9   ]
```

with 3 rows and 4 columns

```
mx set "{{2 3 0.3 0.1} {-4 0.5 4 2} {0.1 -0.2 6 9}}" x
```

is needed. It is generally recommended to use apostrophes so as to prevent Tcl from stripping off the outermost set of curly braces. Below is an example where the contents of a Tcl variable is set to a list {{2 3 4} {5 6 7}} and where this list is passed on to the `mx set` command as an input argument.

```
% set a "{{2 3 4} {5 6 7}}"
{{2 3 4} {5 6 7}}
% mx set $a x
arrayF:0
```

Below is an example where the Tcl -variable x (whose contents was set above) is used as input argument to the `mx puts` command which pretty prints the contents of the matrix to standard output.

```
% mx puts $x
{{2.0 3.0 4.0} {5.0 6.0 7.0}}
% mx puts $x -raw
2.0   3.0   4.0
5.0   6.0   7.0
```

The following example combines the previous two examples to illustrate that formats can be mixed.

```
% set y [mx subtr $x "{{2 3 4} {5 6 7}}"]
arrayF:1
% mx puts $y
{{0.0 0.0 0.0} {0.0 0.0 0.0}}
```

2.6 Accessing a submatrix

Mx allows easy access to a submatrix. Suppose that `x` is the matrix

```
% mx puts $x -raw
2.0   3.0   4.0
5.0   6.0   7.0
```

Then a submatrix of `x` may be specified as a *modification* of `x` by following `x` by a range specification. For example to select elements *1,1* and *1,2*

```
% mx puts $x.(1,1:2)
6.0   7.0
```

As in the above example, when a period follows the variable, it indicates that modifications are to follow. Ranges can be specified in many ways as detailed in the Language Reference. Other modifications such as taking a transpose are also possible.

2.7 Subcommands

Some commands can be applied to either rows or columns. They take a subcommand (`row` or `col`) specifying row or column-wise operation. Examples of such commands include

```
mx sum row $a
mx mean col $b
```

2.8 Controlling and exploiting memory usage

Memory usage forms an integral part of most mathematical packages. One of the powerful features of Mx is that it allows the user to explicitly control memory usage. The user can control whether new memory should be allocated for the result or whether existing memory, if possible, should be reused. Appropriate use can generally lead to considerable algorithmic speed-up[3]. Consider the previous example of adding two matrices

```
% set z [mx add $a $b]
arrayF:2
```

[3]If the idea is to update the contents of an already existing object then it is faster to reuse the memory of this object for the new result. This results in considerable savings of computational time since less low-level memory checking needs to be performed and is generally recommended where possible. If the appropriate object does not yet exist then one has to be created for the result. This involves low-level memory checking and will be slower than reusing memory.

Adding `a` and `b` leads to the result being associated with a new handle `arrayF:2` which was different from the handles for `a` and `b` namely `arrayF:0` and `arrayF:1`. The new handle means that Mx allocated new memory for the result. Suppose that prior to performing the addition the variable `z` did not contain a handle to a matrix object. Then Mx allows the following equivalent syntax for the previous example:

```
% mx add $a $b z
arrayF:2
% puts $z
arrayF:2
```

Here the *name* `z` for the variable that is to hold the result has been specified in *post-fix* position. To Mx a post-fix position for an argument means that it should be treated as an output argument. (Note that since the *contents* of `z` is set the *name* `z` and not the contents `$z` is specified. This is consistent with the Tcl syntax for setting the contents of a variable.)

If however the variable `z` *did* contain a handle to a matrix object, then the two examples need not be equivalent. If `z` contains a handle to a valid matrix object and this object has the right size for the result then Mx will put the result into the memory location used by this matrix object and overwrite the old contents of the matrix. This means Mx does not allocate any new memory for the result, but reuses existing allocated memory. This behavior will be called memory *reusage*. As an example, when `$b` exists,

```
% puts $b
arrayF:1
% mx add $a $b b
arrayF:1
% puts $b
arrayF:1
```

results in the memory associated with `b` being reused and `b` retaining the same handle. On the other hand, if the object did not have the right size Mx destroys it and allocates new memory for it, with behavior similar to the first example.

```
% puts $b
arrayF:1
% mx add $a $b b
arrayF:2
% puts $b
arrayF:2
```

To summarize, a prefix position for the result variable tells Mx to allocate new memory for the result, whereas a suffix position for the result variable (output variable) indicates to Mx to try to reuse the existing memory associated

with that variable if at all possible. If it cannot reuse the memory it destroys the matrix object with the handle given by the output variable, allocates new memory for the result, assigns a handle to this matrix object, and returns it as the contents of the output variable. The next section explains how to destroy objects using the Rtcl package.

2.9 Destroying objects

Mx does not automatically destroy the objects which the user creates as the result of an operation. To preserve memory the user should destroy these objects when they are no longer used. The Rtcl package provides control over memory usage. Table 2.2 lists important Rtcl commands and their use.

Writing and reading objects	`obfile open`, `obfile close`, `obfile read`, `obfile write`, `obfile fields`
Manipulating objects	`object list`, `object nuke`

Table 2.2: Rtcl commands grouped by functionality.

The `object list` command[4] lists all exiting objects. The `object nuke` (or conveniently, simply `nuke`) command destroys objects. In the following example suppose that matrix objects with names `arrayF:0`, `arrayF:1` and `arrayF:2` exist

```
% puts "$a $b $c"
arrayF:0 arrayF:1 arrayF:2
% object list
arrayF:0 arrayF:1 arrayF:2
% object nuke $a $c
% object list
arrayF:1
% nuke $b
% mx add $b $b
object "arrayF:1" does not exist
```

The command

```
nuke [object list]
```

[4]Commands in Rtcl does not have to be preceded by the `rtcl` package name.

destroys all existing objects. As another example of typical use of the `nuke` command

```
set x [mx add $a $b]; nuke $a $b
```

In this example `a` and `b` were no longer needed after the addition operation and were destroyed.

2.10 Writing and reading objects

Another useful command in the Rtcl package is the `obfile` command. This command provides a generic interface for writing and reading objects to disk. Typical use entails

- opening a file, with a handle to the file being returned;
- writing the object to the file with the object associated with a unique key – the field name.
- closing the file.

The `fields` command can be used to obtain a list of the keys of objects already stored in the file. Below follows an example.

```
% mx ones 2 3 x
arrayF:0
% obfile open
wrong # args: should be "obfile open filename ?mode?"
% set filehandle [obfile open filename w]
objfile:1
% obfile write
wrong # args: should be "obfile write filehandle field object"
% obfile write $filehandle myname $x
% obfile fields $filehandle
myname
% set y [obfile read $filehandle myname]
arrayF:2
% obfile close $filehandle
```

Part II

Language Reference

Chapter 3

Syntax

This Chapter gives Mx's syntax and describes objects, different types of row, column, real and imaginary matrices and how to specify them. It also explains how to select ranges of certain elements of these matrices.

3.1 Formal specification

Mx uses a prefix syntax similar to that of Unix and Tcl. Commands composed within the grammar are strings in Tcl and all or any of the tokens, names or symbols may be substituted for with appropriate Tcl variables or arrays. Let ?X? denote that X is *optional* and let | denote the *or* operation. Table 3.1 lists the formal syntax for Mx.

3.2 Objects

In Mx matrix objects are named using a unique text-string *handle* of the form `arrayF:`n, where n is a unique number. This naming is handled entirely by Mx. The `arrayF:`n token is a handle to the n'th matrix object of type `arrayF`. `arrayF` is the name for class *array* objects of type *float*. The handle when returned is guaranteed to be unique for a given object.

3.3 Numbers

Mx accepts numeric input and can return numeric output. A matrix in Mx is a double list, with the inner lists corresponding to rows of the matrix. Note that the double list specification is only necessary to prevent ambiguity. The Tcl-'matrix' token is a valid Tcl list of numbers to specify a matrix. For arbitrary

library command ?subcommand? input ?input? ... ?output? ?options?

library = `mx`
command = `set` | `add` | `puts` | ...
subcommand = `row` | `col` | `min` | `max`

input = *in*?.*mods*?
output = *out*?.*mods*?
in = {*matrices*} | "*matrices*" | *matrix*
out = {*matrices*} | "*matrices*" | *matrix*

options = *options* | `-`*option* ?*args*?

matrices = *matrices* | *matrix*
matrix = name | handle | Tcl-'matrix'
name = string
handle = arrayF:*n*
n = positive integer

mods = ' | (*range*) | '(*range*) | (*range*)'
range = *row*,*col* | *row* | *col*
row | *col* = *beg*:*step*:*end*
beg | *step* | *end* = integer

args = *args* | integer | real | string

arrayF:*n* → refer Section 3.2 Objects
Tcl-'matrix' → refer Section 3.3 Numbers
range → refer Section 3.4 Ranges
.*mods* → refer Section 3.5 Modifiers

Table 3.1: Formal syntax.

numbers `i` and `r`, Table 3.2 gives examples of valid real matrix specifications.

1 x 1 real matrix `r` = `{r}` = `"r"` = `"{r}"` = `"{{r}}"`
1 x n real matrix `"{{r_1 r_2 ... r_n}}"`
m x 1 real matrix `"{{r_1} {r_2} ... {r_m}}"` = `"{r_1 r_2 ... r_m}"`
m x n real matrix `"{{r_11 ... r_1n} {r_21 ... r_2n} ...{r_m1 ... r_mn}}"`

Table 3.2: Real matrices.

For arbitrary numbers `i` and `r`, Table 3.3 gives examples of valid complex matrix specifications.

1 x 1 complex matrix `"r i"` = `"{r} {i}"` = `"{{r}} {{i}}"`
1 x n complex matrix `"{{r_1 r_2 ... r_n}} {{i_1 i_2 ... i_n}}"`
m x 1 complex matrix `"{{r_1} {r_2} ... {r_m}} {{i_1} {i_2} ... {i_m}}"` = `"{r_1 r_2 ... r_m} {i_1 i_2 ... i_m}"`
m x n complex matrix `"{{r_11 ... r_1n} {r_21 ... r_2n} ...{r_m1 ... r_mn}} {{i_11 ... i_1n} {i_21 ... i_2n} ...{i_m1 ... i_mn}}"`

Table 3.3: Complex matrices.

Trying to input complex matrices as `{r} {i}` or `r i` will not work since here `{r} {i}` or `r i` does not describe a complex input matrix but a real input matrix `r` and an output matrix `i`. In this case using `"{r} {i}"` or `"r i"` will work.

3.4 Ranges

In the range specification the tokens *beg*, *step* and *end* are optional (with *beg* defaulting to 0, *step* defaulting to 1 and *end* defaulting to the last row or column depending on its specification with respect to the row column separator `,`. If *step* is omitted then only one separator colon `:` needs to be specified. If the

matrix is a single row or column and both *step* and *beg* or *end* are omitted then no separator colon needs to be specified. 0,0 denotes the left-most, top entry in a matrix.

Here are some examples of valid range modifications

```
$a.(2,:)
$a.(2:4,:)
$a.(2:4,4:)
$a.(2:4,:3)
$a.(2:4,4:5)
```

and

```
set b 2; set e 4
mx puts $a.($b:$e,4:5)
mx add $a.($b:$e,0:1) $a.($b:$e,2:3) a.($b:$e,4:5)
```

Writing

```
$a.(:,:)
```

is legal, but not very efficient. In this case it is better to omit the range specification. Note that for reasons of efficiency the range modifier only allows a step size of 1. For negative step size or step size other than one use for example

```
mx cut $a 2:-1:4,2:2: x
```

3.5 Modifiers

Input and output arguments may be followed by an optional list specifying a modification of the argument. Specifying a range is one example of a modification. The modification list starts with a . symbol (no whitespace) and lists operations that should be performed sequentially on the matrix argument(s) before applying the command (*input*), or before storing the result (*output*).

The modification list is either a (*row*, *column*) range specification (*left*:*step*:*right*,*left*:*step*:*right*) or a ' symbol denoting transposition.

If the input is a list, only the last modification list in the input list is used and this is applied sequentially to all the input arguments in the input list. Complex number arguments are examples of this situation.

Examples where modifiers are used are:

```
set a [mx set "{{1 2 3} {4 5 6} {7 8 9}}"]
mx sqr $a.(1:2,0:1) x
mx log $a.' x
mx cut $a 2:-1:0,0:1:1 x
```

And with complex arguments:

```
mx set "{1 2 3}" br
mx set "{4 5 6}" bi
mx add [list $br [mx set $a.(1:1,0:2)']] \
       [list $bi [mx set $a.(0:0,0:2)']] {xr xi}
set a [mx set "{{1 2 3} {4 5 6} {7 8 9}}"]
mx copy $a b
mx set $a.(1:2,0:1) b.(0:1,1:2);
mx set $a.(1:2,0:1) $b.(0:1,1:2);
set a [mx set "{{1 2} {3 4}} {{5 6} {7 8}}"]
set b [mx copy $a {br bi}]
mx set $a.(1,:) $b.(0,:)
mx set $a.(1,:) {br bi.(0,:)}
mx set $a.(1,:) {$br $bi.(0,:)}
mx sum row $a
mx mean col $br
# done... now destroy the objects
nuke $a $b $br $bi $x $xr $xi
```

Chapter 4

Results

This Chapter explains the different results that Mx returns and discusses the different methods that Mx provides for returning these results. The different types of input and output arguments are explained first because they tend to dictate the type of results. The Chapter also explains how Mx handles scalars.

4.1 Inputs and outputs

The input argument *input* and the optional output argument *output* consist of one or more matrix arguments. Valid matrix arguments include the handle to an existing matrix object. For *input* valid matrix arguments also include a one- or two-dimensional Tcl list of numbers or a scalar while for *output* valid arguments include the name of a Tcl -variable whose contents is a handle to an existing matrix object or can be set to the handle of a newly created matrix object. Table 4.1 provides an algorithmic view of how Mx parses input arguments.

1. **List**
   Check whether *arg* is a list. If so, for each element in the list continue with step 2.
2. **Tcl-'matrix'**
   Check whether *arg* is a valid Tcl list of numbers. If so, then parse them; else
3. **Handle**
   Check whether *arg* is an object handle. If so, then use the object; else return an error message.

Table 4.1: Mx's parsing algorithm for input arguments.

4.2 Passing the result

Mx provides two methods for passing the result.

1. It always returns the result as a text string. This is analogous to passing the result by *value*. (Note that most often the result will be a handle so that passing by value does not constitute a large overhead.)

2. If an output argument is specified it will also return the result (a handle to the matrix object) linked to a Tcl variable of that name where appropriate. This is analogous to passing the result by *reference*.

The next sections explain these two methods in more detail.

4.3 The result as a text string

Mx returns a list of matrix object(s) or a Tcl list of scalar(s). Depending on the command, Mx will do this by returning to the shell (as a text-string) either the handle to the matrix object whose contents is the result, or the numeric result directly in text format. For some commands, instead of just returning the handle of a single matrix object Mx will return a list of handles, one for each of a number of matrix objects. The result of a real scalar operation is the scalar result and not an object.

If the returned result is a Tcl list of numbers (as returned by eg. the `mx puts` command) then it is displayed one row after another. For example
{{2 3 4} {5 6 7}} is a 2 x 3 matrix with 2 rows and 3 columns.
Row one is {2 3 4} and row two is {5 6 7}. Refer also to the `mx set` command.

It is possible to format the result in a number of ways. For example, to get for the result a single matrix element, without {{ }} around it one can use the `puts` with `-raw` option or `value` commands.

```
% mx random 2 1 a
arrayF:1
% foreach x [mx puts $a -raw] {puts "_${x}_"}
_0.396465_
_0.840485_
% mx puts $a.(1) -raw
0.840485
% mx value $a 1
0.840485
```

Note that the `puts -raw` command always inserts an empty line between real and imaginary parts of a matrix even if the imaginary part is absent.

4.4 The result as an output argument

A handle to the result may be passed to Tcl-level variables through the optional output argument. The parsing algorithm for the output argument is described below and listed in Table 4.2.

If no *output* argument is specified then Mx will always use new memory and create an appropriately sized matrix object for the output. If instead an *output* argument is specified then Mx will try to reuse memory as follows.

- If the *output* argument is a handle to an existing matrix object or it is the name of a Tcl variable whose contents is a handle to an existing matrix object and, this object is appropriately sized for the result, then Mx overwrites those of its existing elements that pertain to the result. (For example if the result pertains to a subrange of the output matrix then only elements of that subrange will be overwritten. This is useful to update a single or subset of elements in the matrix while not affecting the other elements.)

- If the *output* argument is a name or string that does not refer to an existing matrix object or this object is not appropriately sized for the result then Mx allocates new memory for an appropriately sized object, sets its contents and updates the Tcl-variable with the same name as the output argument and sets its contents to the handle of the matrix object.

Table 4.2 provides an algorithmic view of how Mx parses output arguments.

1. **List**
   Check whether *arg* is a list. If so, for each element in the list continue with step 2.

2. **Handle**
   Check whether *arg* is an object handle. If so, then try to reuse the object; else

3. **Variable**
   Check whether *arg* is a Tcl variable. If so, then get its contents, name it *arg* and continue with step 1; else

4. **Name**
   Create a Tcl variable with the name *arg* and make its contents the handle to the new object.

Table 4.2: Mx's parsing algorithm for output arguments.

4.5 Scalars

For scalar operations Mx is similar to Tcl 's `expr` command as long as a stack is not needed for parsing. The example below explains how to multiply or divide by a scalar.

```
mx set "{{2 3 4 5.5}}" a
set b [mx mul $a 2]
set d 2.3
set c [mx div $a $d]
set c [mx mul 2.3 4.5]
```

Note that Mx is designed to *not* return an object if you input only non-object scalars. This is due to parsing efficiency considerations since it avoids an additional unnecessary object look-up and memory allocation. In the example

```
mx mul $a.(2:3) 4.8 a.(2:3)
```

Mx will multiply elements 2 and 3 (counting from 0) by 4 and leave the rest of `a` intact, with the result again available in `a`.

All of the above vector operations generalize to matrix operations as well so it is valid to do

```
mx ones 10 20 b
set c [mx mul 42 $b]
```

which will multiply the whole matrix element-wise with 42. A further example is

```
set d [mx mul $b.(3:4,5:) 7.1].
```

Chapter 5

Messages

This Chapter describes Mx's error and exception handling capabilities. Although the exception handling is fairly rudimentary, it does allow for fast execution with reasonable verbosity. Future versions of Mx are likely to have more advanced error and exception handling.

5.1 Errors

If Mx detects a syntax error it tries to return an informative message. If a script was executed Tcl will return a Tcl command trace at the point of error.

```
% mx add $x
wrong # of args: should be "mx add inobjA inobjB ?outobj?"
% mx add x x
expected floating-point number but got "x"
```

5.2 Exceptions

Mx provides only rudimentary exception checking. Generally a floating point exception will return an error during an interactive session or execution of a script. With some mathematical libraries a command such as

```
mx div $x 0 y
```

may simply return the IEEE arithmetic result

```
% mx puts $y
{{Inf Inf} {Inf Inf}}
```

and not generate an exception.

Part III

Command Reference

Chapter 6

Matrix manipulation

This Chapter provides a complete reference for all the commands available in Mx for manipulating matrices such as selecting elements in a certain range of a matrix or transposing a matrix. It explains each command, describes its arguments and options and gives examples of use.

6.1 Synopsis

mx copy *a* ?*z*?
mx cut *a range* ?*z*?
mx join col "{*a b ...*}" ?*z*?
mx join row "{*a b ...*}" ?*z*?
mx join col [list *a b ...*] ?*z*?
mx join row [list *a b ...*] ?*z*?
mx size *a* ?*z*?
mx dim *a* ?*z*? ?**-row**? ?**-col**? ?**-last**?
mx ones *dim1 dim2* ?*z*?
mx zeros *dim1 dim2*
mx eye *dim1 dim2*
mx random *dim1 dim2* ?*z* ?**-seed** *seed*? ?**-int**? ?**-range** *from to*?
mx diag *a* ?*z*?
mx zero *a* ?*z*?
mx one *a* ?*z*?
mx zerodiag *a* ?*z*?
mx onediag *a* ?*z*?
mx scale row *a b* ?*z*?
mx scale col *a b* ?*z*?
mx weigh *a* ?*z*? ?**-exp** *num*?
mx tr *a* ?*z*?
mx unwrap *a* ?*z*?

mx flipud *a* ?*z*?
mx fliplr *a* ?*z*?
mx load constants
mx unload constants

6.2 Commands

mx copy *a* ?*z*?
Copy *a*.

mx cut *a range* ?*z*?
Cut block from *a*. (0,0 is the index of the first element). The *range* argument can take any of the following forms:

range

i:*j*:*k*,*l*:*m*:*n*
select every j'th row from row i to k, and every m'th column from column l to n

i:*j*
select elements *i* to *j*

i:*j*
is empty if $i > j$

i:*s*:*j*
select elements *i* to *j* skipping in steps of *s*

i:*s*:*j*
is empty if $s > 0$ and $i > j$ or if $s < 0$ and $i < j$

Example

```
set a [mx set "{{1 2 3 4 5 6} {1 2 3 4 5 6} {1 2 3 4 5 6}}"]
mx cut $a 0:1,1:2:5 z
set z [mx cut $a:,3:-1:0]
```

mx join col "{*a b* ...}" ?*z*?

mx join row "{*a b* ...}" ?*z*?

mx join col [list *a b* ...] ?*z*?

mx join row [list *a b* ...] ?*z*?
Join one or more matrices or vectors *a* column- or row-wise. The input argument can be a Tcl list of matrices or vectors to be joined.

Example

```
set a [list $a1 $a2 $a3]
mx join col $a z
set z [mx join row "{$a1 $a2}"]
set z [mx join row [list $a $z]]
```

mx size *a ?z?*
Size of matrix or vector *a*. The result is a row vector {{rows columns}}.

mx dim *a ?z? ?***-row***? ?***-col***? ?***-last***?*
Size of matrix or vector *a*.

-row
return only the row dimension

-col
return only the column dimension

-last
return the index of last row and/or column instead of the dimension

Example

```
puts [mx dim -row $a z]
mx puts $z
set z [mx dim $a]
set rows [lindex $z 0];  set cols [lindex $z 1]
puts [mx dim -last $a z]
```

Returns

Tcl list
If no option is specified the result is a row vector {{rowspec colspec}} else with **-row** the result is {rowspec} or with **-col** {colspec}.

Vector object
A vector object with the size in *z* if this is specified.

mx ones *dim1 dim2 ?z?*
Create a *dim1* by *dim2* matrix with all elements 1.0.

Example

```
mx ones 1 2 z
```

mx zeros *dim1 dim2*
Create a *dim1* by *dim2* matrix with all elements 0.0.

Restrictions
Do not use this command with memory reusage - results will be indeterminate.

Example

```
set z [mx zeros {2 2}]
```

mx eye *dim1 dim2*
Create the identity matrix; elements on the diagonal 1.0; other elements 0.0.

Restrictions
Do not use this command with memory reusage - results will be indeterminate.

Example

```
set z [mx eye 2 2]
```

mx random *dim1 dim2* ?*z*? ?**-seed** *seed*? ?**-int**? ?**-range** *from to*?
Create a *dim1* by *dim2* matrix whose elements are pseudorandom numbers generated using either the drand48, random or rand library functions. (Courtesy of Jacques de Villiers.)

-seed
integer seed value, for example Tcl 's [**clock seconds**] (default 0)

-int
generate random integers (floats by default)

from to
(from,to) for floats and [from,to) for integers. Default values are (0,1) for floats and [0,2147483648) for integers.

Example

```
mx load constants
# returns a 5 by 10 matrix of random
# real numbers between zero and pi
mx random 5 10 z -seed 42 -range 0 $mxpi
mx random 5 10 z -seed [clock seconds] -range 0 $mxpi
```

mx diag *a* ?*z*?
Convert diagonal of matrix *a* to row vector or vector to diagonal of matrix.

mx zero *a* ?*z*?
Make all elements in *a* zero.

mx one *a* ?*z*?
Make all elements in *a* one.

mx zerodiag *a* ?*z*?
Make all elements on diagonal of matrix *a* zero.

mx onediag *a* ?*z*?
Make all elements on diagonal of matrix *a* one.

mx scale row *a b* ?*z*?

mx scale col *a b ?z?*
Multiply each row or column in matrix *a* by the corresponding element in vector *b*. A typical application is fast matrix-matrix multiplication if one matrix is diagonal.

a
input vector to be scaled (m x n)

b
input scaling vector (must be 1-dimensional and length n for **col**, length m for **row**)

Example

```
mx scale row $a $b
set z [mx scale row $a $b]
mx scale col $a $b z
```

mx weigh *a ?z? ?-exp num?*
Multiplies each column i of a by the weight $w(i) = i^{num}$.

-exp *num*
Weighing exponent. [default, 0]

Example

```
mx weigh $A -exp 1
set X [mx weigh $A -exp 2]
```

mx tr *a ?z?*
Matrix or vector transpose of *a*.

mx unwrap *a ?z?*
Concatenate rows of matrix *a* into one row vector.

mx flipud *a ?z?*
Flip columns of *a* in up-down direction.

mx fliplr *a ?z?*
Flip rows of *a* in left-right direction.

(Modifier) **'**
Matrix or vector transpose.

(Modifier) **(***range***)**

range

i:*j*:*k*,*l*:*m*:*n*
select every j'th row from row i to k, and every m'th column from column l to n

i:*j*
select elements *i* to *j*

i:*j*
: is empty if $i > j$

i:*s*:*j*
: select elements i to j skipping in steps of s

i:*s*:*j*
: is empty if $s > 0$ and $i > j$ or if $s < 0$ and $i < j$

mx load constants

mx unload constants

Loads (or unloads and frees) predefined constants such as PI in memory.

mxpi
: 3.1415926535897932385 = pi = [mx mul 4 [mx atan 1]]

mxpi2
: 6.3661977236758134308 = 2pi

mxpi_2
: 1.5707963267948966192 = pi/2

mxsqrt2
: 1.4142135623730950488 = sqrt 2

Returns
: Zero or more vector objects.

Example

```
mx load constants
mx unload constants
mx puts $mxpi
```

Chapter 7

Input and output

This Chapter provides a complete reference for all the commands available in Mx for input and output of matrices, including the different formats and explaining each command, describing its arguments and options and giving examples of use.

7.1 Synopsis

mx set *a* ?*z*?
mx puts *a* ?**-raw**? ?**-text**? ?**-nonewline**? ?**-int**? ?**-precision** *int*?
mx value *a i j* ?**-int**? ?**-precision** *int*?
mx fprintf *filename a* ?**-format** *string*? ?**-append**? ?**-header**?
mx fprintf *filename a* ?**-pre** *string*? ?**-post** *string*? ?**-preline** *string*?
?**-postline** *string*?
mx fscanf *filename* ?*z*? ?**-c** *int*? ?**-r** *int*?
mx fwrite *filename a* ?**-header**? ?**-append**?
mx fread *filename* ?*z*? ?**-c** *int*? ?**-r** *int*?

7.2 Commands

mx set *a* ?*z*?
Set elements in a matrix.

a
The handle to an existing matrix object (the name of an arrayF-type object) or a *Tcl-matrix.* or a scalar.

z
The handle to an existing matrix object (the name of an arrayF-type

object) or the name of a Tcl variable whose contents is a handle to an existing matrix object or can be set to the handle of a newly created matrix object.

Tcl-matrix

1 x 1 real matrix
r = {r} = "r" = "{r}" = "{{r}}"

1 x n real matrix
"{{r_1 r_2 ... r_n}}"

m x 1 real matrix
"{{r_1} {r_2} ... {r_m}}" = "{r_1 r_2 ... r_m}"

m x n real matrix
"{{r_11 ... r_1n} {r_21 ... r_2n} ...{r_m1 ... r_mn}}"

1 x 1 complex matrix
"r i" = "{r} {i}" = "{{r}} {{i}}"

1 x n complex matrix
"{{r_1 r_2 ... r_n}} {{i_1 i_2 ... i_n}}"

m x 1 complex matrix
"{{r_1} {r_2} ... {r_m}} {{i_1} {i_2} ... {i_m}}" = "{r_1 r_2 ... r_m} {i_1 i_2 ... i_m}"

m x n complex matrix
"{{r_11 ... r_1n} {r_21 ... r_2n} ...{r_m1 ... r_mn}} {{i_11 ... i_1n} {i_21 ... i_2n} ...{i_m1 ... i_mn}}"

Trying to input complex matrices as {r} {i} or r i will not work since here {r} {i} or r i does not describe a complex input matrix but a real input matrix 'r' and an output matrix 'i'. In this case using "{r} {i}" or "r i" will work.

Examples

```
mx set "{{1 2.3} {4.5 5}}" z
set z [mx set "{-0.2 0}"]
mx set "{{1 2.3} {4.5 5}} {{-2 3} {.5 5}}" {zr zi}
set z [mx set "-2.0 -2.0"]
# This makes a copy of a
set z [mx set $a]
# This tries to reuse z
mx set $a z
# This makes an alias z of a
set z $a
```

mx puts *a* ?**-raw**? ?**-text**? ?**-nonewline**? ?**-int**? ?**-precision** *int*?
Take the input and returns a Tcl list or formatted text.

-raw
Output elements as raw numbers, column aligned. [Tcl list]

-text
Output elements as a text block. [off]

-nonewline
Do not print newline after every row (requires **-text**). [off]

-int
Convert numbers to type integer by truncating decimal part.

-precision *int*
Use this precision for output. [$tcl_precision] The **-raw** option must be given as well.

Examples

```
mx puts $ar
set z [mx puts $ar]
puts [mx puts [list $ar $ar]]
```

Returns
Tcl list or text.

mx value *a i j* ?**-int**? ?**-precision** *int*?
Returns the contents (numeric value) of element *i,j* in *a* as a text string.

-int
Convert numbers to type integer by truncating decimal part.

-precision *int*
Use this precision for output. [$tcl_precision]

Examples

```
mx value $a 2 5
set z [mx value $a -int 3 4]
```

Returns
A text string.

mx fprintf *filename a* ?**-format** *string*? ?**-append**? ?**-header**?

mx fprintf *filename a* ?**-pre** *string*? ?**-post** *string*? ?**-preline** *string*? ?**-postline** *string*?
Write matrix in ASCII format to *filename*. A header of the form
<rows[*int*]> <columns[*int*]>
is written if the **-header** flag is set. If *filename* is stdout then output is written to stdout. If *filename* is stderr then output is written to stderr.

-format *string*
Use this c-type format string. ["%f "]

-append
Append to file. [do not append]

-header
Write header. [don't write header]

-pre *string*
Prepend output with *string*.

-post *string*
Append output with *string*.

-preline *string*
Prepend each line with *string*.

-postline *string*
Append each line with *string*.

Returns
Success - number of lines and columns written.

Restrictions
This is a fairly inefficient writing routine and should be used mainly for small tasks and debugging purposes.

Example

```
mx fprintf file1 $a
set z [mx fprintf "file2" $a -format "%2.3f " -header -append]
```

mx fscanf *filename* ?*z*? ?**-c** *int*? ?**-r** *int*?
Read matrix in ASCII format from *filename*. If a header of the form
<rows[*int*]> <columns[*int*]>
is found it is read and used unless overridden by the **-c** or **-r** flags. If no header is found and no flag specified then a row vector is read. If *filename* is stdout then input is read from stdout. If *filename* is stderr then input is read from stderr.

-c *int*
Read these many columns. [number of columns from header]

-r *int*
Read these many rows. [number of rows from header]

Restrictions
This is a fairly inefficient read routine and should be used mainly for small tasks and debugging purposes.

Example

```
mx fscanf file1 z
set z [mx fscanf "file2" -r 10 -c 5]
```

mx fwrite *filename a* ?**-header**? ?**-append**?
Write matrix in binary format to file. A header of the form
<rows[*int*]> <columns[*int*]>
is written if the **-header** flag is set.

-header
Write header. [don't write header]

-append
Append to file. [do not append]

Restrictions
Data is written in host byte order.

Example

```
mx fwrite file1 $a
set z [mx fwrite file2 $a]
```

mx fread *filename ?z? ?***-c** *int? ?***-r** *int?*
Read matrix in binary format from file. If a header of the form
<rows[*int*]> <columns[*int*]>
is found it is read and used unless overridden by the **-c** or **-r** flags. If no header is found and no flag specified then a row vector is read.

-c *int*
Read these many columns. [number of columns from header]

-r *int*
Read these many rows. [number of rows from header]

Restrictions
Data is read in host byte order.

Example

```
mx fread file1 z
set z [mx fread file2 -r 10 -c 5]
```

Chapter 8

Basic mathematics

This Chapter provides a complete reference for all the commands available in Mx for performing basic mathematics explaining each command and describing its arguments.

8.1 Synopsis

mx trace *a ?z?*
mx prod *a b ?z?*
mx mul *a b ?z?*
mx div *a b ?z?*
mx rem *a b ?z?*
mx add *a b ?z?*
mx subtr *a b ?z?*
mx sqr *a ?z?*
mx abs *a ?z?*
mx angle *a ?z?*
mx real *a ?z?*
mx imag *a ?z?*
mx conj *a ?z?*
mx c2p *a ?z?*
mx p2c *a ?z?*
mx pmul *a b ?z?*
mx pdiv *a b ?z?*
mx padd *a b ?z?*
mx psubtr *a b ?z?*
mx psqr *a ?z?*
mx pabs *a ?z?*
mx pangle *a ?z?*
mx preal *a ?z?*

mx pimag *a ?z?*
mx ceil *a ?z?*
mx floor *a ?z?*
mx sign *a ?z?*
mx cos *a ?z?*
mx sin *a ?z?*
mx tan *a ?z?*
mx acos *a ?z?*
mx asin *a ?z?*
mx atan *a ?z?*
mx cosh *a ?z?*
mx sinh *a ?z?*
mx tanh *a ?z?*
mx sqr *a ?z?*
mx sqrt *a ?z?*
mx exp *a ?z?*
mx log *a ?z?*
mx log10 *a ?z?*

8.2 Commands

mx trace *a ?z?*
Sum of matrix elements along diagonal.

mx prod *a b ?z?*
Inner-product matrix multiplication *a*b*.

Restrictions
The row dimension of *a* and column dimension of *b* must be the same.

mx mul *a b ?z?*
Multiply each element in *a* with the corresponding element in *b*.

mx div *a b ?z?*
Divide each element in *a* by the corresponding element in *b*.

mx rem *a b ?z?*
The remainder after dividing each element in *a* by the corresponding element in *b*.

mx add *a b ?z?*
Add each element in *a* to the corresponding element in *b*.

mx subtr *a b ?z?*
Subtract each element in *b* from the corresponding element in *a*.

mx sqr *a ?z?*
Multiply each element in *a* with itself.

mx abs *a ?z?*
Absolute value of the elements of *a*.

mx angle *a ?z?*
Phase angle in radians of the elements of *a*.

mx real *a ?z?*
Complex real part of the elements of *a*.

mx imag *a ?z?*
Complex imaginary part of the elements of *a*.

mx conj *a ?z?*
Complex conjugation of the elements of *a*.

mx c2p *a ?z?*
Cartesian to polar transformation of the elements of *a*.

mx p2c *a ?z?*
Polar to cartesian transformation of the elements of *a*.

(Polar coordinates)

mx pmul *a b ?z?*
Multiply each element in *a* with the corresponding element in *b*, given input and output in polar coordinates.

mx pdiv *a b ?z?*
Divide each element in *a* by the corresponding element in *b*, given input and output in polar coordinates.

mx padd *a b ?z?*
Add each element in *a* to the corresponding element in *b*, given input and output in polar coordinates.

mx psubtr *a b ?z?*
Subtract each element in *b* from the corresponding element in *a*, given input and output in polar coordinates.

mx psqr *a ?z?*
Multiply each element in *a* with itself, given input and output in polar coordinates.

mx pabs *a ?z?*
Return absolute value of its elements, given input and output in polar coordinates.

mx pangle *a ?z?*
Return phase angle in radians, given input and output in polar coordinates.

mx preal *a ?z?*
Return complex real part, given input and output in polar coordinates.

mx pimag *a ?z?*
Return complex imaginary part, given input and output in polar coordinates.

(Numeric)

mx ceil *a ?z?*
Round elements of *a* towards plus infinity.

mx floor *a ?z?*
Round elements of *a* towards minus infinity.

mx sign *a ?z?*
Signum function of elements of *a*.

(Trigonometric)

mx cos *a ?z?*
Cosine of elements of *a*.

mx sin *a ?z?*
Sine of elements of *a*.

mx tan *a ?z?*
Tangent of elements of *a*.

mx acos *a ?z?*
Arc cosine of elements of *a*.

mx asin *a ?z?*
Arc sine of elements of *a*.

mx atan *a ?z?*
Arc tangent of elements of *a*.

mx cosh *a ?z?*
Hyperbolic cosine of elements of *a*.

mx sinh *a ?z?*
Hyperbolic sine of elements of *a*.

mx tanh *a ?z?*
Hyperbolic tangent of elements of *a*.

(Exponential)

mx sqr *a ?z?*
Multiply each element of *a* with itself.

mx sqrt *a ?z?*
Square root of each element of *a*.

mx exp *a ?z?*
The exponential of each element of *a*.

mx log *a ?z?*
The natural logarithm of each element of *a*.

mx log10 *a ?z?*
The base 10 logarithm of each element of *a*.

Chapter 9

Decompositions and transformations

This Chapter provides a complete reference for all the commands available in Mx for performing decompositions and transformations of the data such as Cholesky decompositions or Fast Fourier Transforms. It explains each command, describes its arguments and options and gives examples of use.

9.1 Synopsis

mx chol *a* ?*z*?
mx cholinv *a* ?*z*?
mx cholsolve *a b* ?*z*?
mx fft *a* ?*z*?
mx ifft *a* ?*z*?
mx norm *a b* ?**-l** *num*?
mx normalize *a* ?**-l** *num*?
mx qsort ?**-increasing**? *a* ?*z*?
mx qsort ?**-decreasing**? *a* ?*z*?

9.2 Commands

mx chol *a* ?*z*?
: Cholesky decomposition of *a* into *z***z*'. *z* is lower triangular. A typical application is to get a matrix square root.

a
: Positive definite m by m input vector.

z
: Optional m by m output vector.

Example

```
mx chol $a z
set z [mx chol $a]
```

mx cholinv *a ?z?*
: Inverse of a by Cholesky decomposition. A typical application is to get the inverse of a covariance matrix.

a
: Positive definite m by m input vector.

z
: Optional m by m output vector.

Example

```
mx cholinv $a z
set a [mx cholinv $z]
```

mx cholsolve *a b ?z?*
: Solution z by Cholesky decomposition of $a * z = b$. Symmetry of a is not required - only the upper triangle of a is used.

a
: Positive definite m by m input vector.

b
: m by n vector.

z
: Optional m by n output vector for the solution.

Example

```
mx cholinv $a $b z
set z [mx cholinv $a $b]
```

mx fft *a ?z?*

mx ifft *a ?z?*
: Real or complex Fast Fourier Transform (row-wise) or inverse Fast Fourier Transform (row-wise). The input vector a is padded with zeros up to the nearest num=power-of-two length. The inverse FFT is calculated by doing conj(fft(conj(a)))/N. Both real and complex FFTs are supported, the complex case being about 10 percent slower than the real case.

Returns
A two element list of vector objects (real and imaginary components).

Example

```
mx fft "{$ar $ai}" {zr zi}
set a [list $ar $ai]
set z [mx fft $a]
set zr [lindex $z 0]; set zi [lindex $z 1]
mx ifft "{$zr $zi}" {ar ai}
set a [mx ifft $z]; nuke $z
set ar [lindex $a 0]; set ai [lindex $a 1]
```

mx norm *a b* ?**-l** *num*?
l-norm of one or difference of two row vector sequences. Calculates the l=*num* norm for each column vector of *a* (if only *a* is specified this is ||*a*||) or if both *a* and *b* are specified this is ||*a*-*b*||). Averages all the resulting norms over the sequence and returns the resulting number. By default the l=2 (Euclidean) norm is calculated. Valid l-norms are l=1, l=2, l=**inf** and l=**mininf**, and l equal to any positive non-zero real. If *b* is not specified and the sequence length is 1 the command returns the conventional l-norm of the input vector.

-l *num*
use l=*num* norm [default 2, the Euclidean norm]

Returns
The norm as a single float number.

Example

```
mx norm -l inf $a $b
set z [mx norm -l 2 $a $b]
set z [mx norm -l 2 $a]
```

mx normalize *a* ?**-l** *num*?
Normalize each row vector of the input vector *a* by given l=*num* norm. By default the l=2 (Euclidean) norm is calculated. Valid l-norms are l=1, l=2, l=**inf** and l=**mininf**, and l equal to any positive non-zero real.

-l *num*
use l=*num* norm [default 2, the Euclidean norm]

Example

```
mx normalize $a z
set z [mx normalize $a z -l inf]
```

mx qsort ?**-increasing**? *a* ?*z*?

mx qsort ?**-decreasing**? *a* ?*z*?
Sort elements in each row of *a* using the quick sort algorithm.

-increasing
sort in increasing order [on]

-decreasing
sort in decreasing order [off]

Returns
A vector object of the sorted input.

Example

```
mx qsort $a z
set z [mx qsort -increasing $a]
```

Chapter 10

Elementary statistics

This Chapter provides a complete reference for all the commands available in Mx for performing elementary statistical operations such as getting the mean or variance of a sample. It explains each command, describes its arguments and options and gives examples of use.

10.1 Synopsis

mx find max *a ?z?*
mx find min *a ?z?*
mx cov *a ?z?*
mx corr *a ?z?*
mx mean row *a ?z?*
mx mean col *a ?z?*
mx min row *a ?z?*
mx min col *a ?z?*
mx max row *a ?z?*
mx max col *a ?z?*
mx sum row *a ?z?*
mx sum col *a ?z?*
mx std row *a ?z?*
mx std col *a ?z?*
mx zeromean *a ?z?*

10.2 Commands

mx find max *a ?z?*

mx find min *a* ?*z*?
Find index and value of largest or smallest element.

Returns

Tcl list
Three number Tcl string of {{rowindex colindex value}}.
A string of elements describing the index and particular value.

Vector object
A vector object with index and value in *z* if this is specified.

Example

```
mx min $a z
mx puts $z
set z [mx find min a]
set index [lindex $z 0]
set value(index) [lindex $z 1]
```

mx cov *a* ?*z*?
Covariance matrix of *a* (row major order). The result is *a*'**a*/n - mean(*a*)'*mean(*a*), where n is the number of rows in *a*.

Example

```
mx cov $a z
set z [mx cov $a]
```

mx corr *a* ?*z*?
Correlation matrix (energy) of *a* (row major order). The result is *a*'**a*/n, where n is the number of rows in *a*.

Example

```
mx corr $a z
set z [mx corr $a]
```

mx mean row *a* ?*z*?

mx mean col *a* ?*z*?
Mean of elements in each row or column of *a*.

mx min row *a* ?*z*?

mx min col *a* ?*z*?
Index of smallest element in each row or column of *a*.

mx max row *a* ?*z*?

mx max col *a* ?*z*?
Index of largest element in each row or column of *a*.

mx sum row *a ?z?*

mx sum col *a ?z?*
Sum of the elements in each row or column of *a*.

mx std row *a ?z?*

mx std col *a ?z?*
Sample standard deviation of elements in each row or column of *a*.

mx zeromean *a ?z?*
Subtract the mean from each column of *a*.

Part IV

Appendix

Appendix A

Availability

Mx acts as a package extension of the Tcl script language [1]. Currently Mx is bundled with the *Center for Spoken Language Understanding shell* [6]. Mx is available as part of the CSLU toolkit free of charge for academic and research purposes at `http://www.cse.ogi.edu/CSLU/toolkit/`. Before running Mx it is necessary to have the toolkit installed on your system.

Appendix B

Examples

A hands-on approach is often the best way of learning how to use a new tool. This Chapter provides a number of simple examples detailing how some of Mx's features and capabilities may be put to use.

Mx is used extensively at the Center for Spoken Language Understanding and in the Anthropic Speech Processing Group at the Oregon Graduate Institute of Science and Technology. It has been used for projects such as speech recognition [2] and speaker verification [3, 4]. Mx provides a rich variety of uses. The following examples demonstrates just a few of these uses.

B.1 One line examples

Some simple real matrix examples of using Mx are

```
% mx set "{{-1.0  2.0}}" a
arrayF:0
% set x [mx sqrt $a.(:,1)]
arrayF:1
% mx puts $x
{{1.41421}}
% mx puts [mx add $a.'  $a.' x]
{{-2.0} {4.0}}
% nuke $a $x
```

Complex matrices are supported as a doublet of matrices. Below are examples.

```
set a [mx set "{{3.1 2 2}} {{5 -6 7}}"]
set b [mx set "{2 1 1} {3 3 4}" {br bi}]
set x [mx prod $a $b {xr xi}]
set x [mx prod $a [list $br $bi] {xr xi}]
```

```
puts [mx puts $x]
puts [mx puts [list $xi $xr]]
```

B.2 Associativity

The marriage of Mx and Tcl allows for associative matrices. For example for speaker identification on the TIMIT corpus, one may built up a confusion matrix using

```
mx set $value conf($region,$caller).($i,$j)
mx puts $conf(dr2,cjf0).(:,2)
mx puts $conf(dr4,dma0).(6,6)
```

B.3 Data manipulation

This example demonstrates Mx's `join` command where the application is a speech recognition system using a neural-net to estimate aposteriori probabilities. The idea is to 'collapse' the output `nn` from a neural-net, that is combine as their average certain columns in the 'probability' matrix. First it is necessary to generate a makeshift neural-net output for this example

```
mx prod [mx ones 3 1 t] "{{0 1 2 3 4 5 6 7}}" nn; nuke $t
mx puts $nn -raw
```

This gives the index of the last column in `nn`

```
set n [mx dim -col -last $nn]
```

Here is the collapse specification

```
set collapse "nc 0 2 c 3 4 nc 5 5 c 6 $n"
```

This does the actual collapse of columns

```
set lst {}
foreach {type b e} $collapse {
 switch $type {
  c  {lappend lst [mx mean row $nn.(:,$b:$e)]}
  nc {lappend lst [mx copy $nn.(:,$b:$e)]}
 }
}
set nn_new [mx join col $lst]; nuke $lst
```

Finally this outputs the result

```
mx puts $nn_new -raw
```

B.4 Arithmetic Harmonic Sphericity Measure

Consider implementing the arithmetic harmonic sphericity measure (AHS) [8]

$$\log[\mathrm{tr}(\mathrm{cov}(Y) * \mathrm{cov}(X)^{-1}) * \mathrm{tr}(\mathrm{cov}(X) * \mathrm{cov}(Y)^{-1})] - 2 * \log(d).$$

Here is a procedure to calculate the AHS measure

```
proc ahs {X Y} {
  mx cov $X Sx
  mx cov $Y Sy
  mx cholinv $Sx Si
  set tx [mx value [mx trace [mx prod $Sy $Si p] t] 0]
  mx cholinv $Sy Si
  set ty [mx value [mx trace [mx prod $Sx $Si p] t] 0]
  nuke $Sx $Sy $Si $p $t
  expr log($tx*$ty)-2*log([mx dim -col $X])
}
```

An example call is

```
% mx random 4 2 x
% mx random 4 2 y
% ahs $x $y
2.15865
```

Procedure `ahs` has a fair number of temporary variables. Now suppose that it is known that the procedure will be called many times. Then it may be advantageous to allocate memory space for the variables only once, that is enforce an indefinite existence with only their contents left temporary. This enforces *persistency* of the temporary variables. Note of that it is the *existence* and not necessarily the *contents* of the temporary variables that is persistent. Here is an implementation exploiting this idea.

```
proc ahs2 {X Y} {
  global Sx Sy Si p t
  mx cov $X Sx
  mx cov $Y Sy
  mx cholinv $Sx Si
  set tx [mx value [mx trace [mx prod $Sy $Si p] t] 0]
  mx cholinv $Sy Si
  set ty [mx value [mx trace [mx prod $Sx $Si p] t] 0]
  expr log($tx*$ty)-2*log([mx dim -col $X])
}
```

When called many times, procedure `ahs2` will be considerably faster than procedure `ahs` due to its smaller memory management overhead.

Appendix C

Algorithms

This Appendix contains a concise listing of all the commands available in Mx. For each command it specifies input and output types for the arguments. It also gives an example of Mx code for the command as well as pseudo-code.

abs

Args:(in) : real complex

Args:(out) : real

Code:(pseudo): z=abs(a);

acos

Args:(in,out): real

Code:(pseudo): u=abs(a)<=1; z=acos(a.*u+ones(size(a)).*(1-u));

add

Args:(in,out): real mixed complex

Code:(pseudo): z=a+b;

angle

Args:(in) : real complex

Args:(out) : real

Code:(pseudo): z=angle(a);

asin

Args:(in,out): real

Code:(pseudo): u=abs(a)<=1; z=asin(a.*u);

atan

Args:(in,out): real
Code:(pseudo): z=atan(a);

c2p

Args:(in,out): real mixed
Code:(pseudo): z=abs(a)+im*angle(a);

ceil

Args:(in,out): real mixed
Code:(pseudo): z=ceil(a);

chol

Args:(in,out): real
Code:(mx) : mx chol $d z
Code:(pseudo): z=chol(d)';

cholinv

Args:(in,out): real
Code:(mx) : mx cholinv $d z
Code:(pseudo): u=chol(d); z=inv(u)*inv(u');

cholsolve

Args:(in,out): real
Code:(mx) : mx cholsolve $d $c z
Code:(pseudo): z=inv(d)*c;

conj

Args:(in,out): real mixed
Code:(pseudo): z=conj(a);

copy

Args:(in,out): real mixed
Code:(pseudo): z=a;

corr

Args:(in,out): real
Code:(mx) : mx corr $a z
Code:(pseudo): [sz sz1]=size(a); z=a'*a/sz;

cos

Args:(in,out): real
Code:(pseudo): z=cos(a);

cosh

Args:(in,out): real
Code:(pseudo): z=cosh(a);

cov

Args:(in,out): real
Code:(mx) : mx cov $a z
Code:(pseudo): z=cov(a);

cut

Args:(in,out): real
Code:(mx) : mx cut $a 1:2:4,0:2 z
Code:(pseudo): z=a(2:2:5,1:3);

cut

Args:(in,out): real
Code:(mx) : mx cut $a 0,1 z
Code:(pseudo): z=a(1,2);

cut

Args:(in,out): real
Code:(mx) : mx cut $a :,2:-1:0 z
Code:(pseudo): z=a(:,3:-1:1);

cut

Args:(in,out): real
Code:(mx) : mx cut $a 0,: z
Code:(pseudo): z=a(1,:);

diag

Args:(in,out): real mixed
Code:(pseudo): z=conj(diag(a)');

dim

Args:(in,out): real

Code:(mx) : mx dim $a z
Code:(pseudo): z=size(a);

dim

Args:(in,out): real
Code:(mx) : mx dim -row $a z
Code:(pseudo): [z u]=size(a);

dim

Args:(in,out): real
Code:(mx) : mx dim -col $a z
Code:(pseudo): [u z]=size(a);

dim

Args:(in,out): real
Code:(mx) : mx dim -row -last $a z
Code:(pseudo): u=size(a)-[1 1];z=u(1,1);

dim

Args:(in,out): real
Code:(mx) : mx dim -col -last $a z
Code:(pseudo): u=size(a)-[1 1];z=u(1,2);

div

Args:(in,out): real mixed complex
Code:(pseudo): z=a./b;

exp

Args:(in,out): real
Code:(pseudo): z=exp(a);

eye

Args:(in,out): real
Code:(mx) : mx eye 2 3 z
Code:(pseudo): z=eye(2,3);

eye

Args:(in,out): real
Code:(mx) : mx eye 1 4 z

Code:(pseudo): z=eye(1,4);

eye

Args:(in,out): real

Code:(mx) : mx eye 3 1 z

Code:(pseudo): z=eye(3,1);

fft

Args:(in,out): real complex

Code:(mx) : set z [mx fft "$a $a"]

Code:(pseudo): [sz sz1]=size(a); if sz*sz1~=sz, u=conj((a+im*a)'); z=conj(fft(u)'); else z=a+im*a; end

fft

Args:(in,out): real complex

Code:(mx) : set z [mx fft $a]

Code:(pseudo): [sz sz1]=size(a); if sz*sz1~=sz, u=a'; z=conj(fft(u)'); else z=a; end

find

Args:(in,out): real

Code:(mx) : mx find min $a u; mx set $u.(2) z; nuke $u

Code:(pseudo): z=min(min(a));

find

Args:(in,out): real

Code:(mx) : mx find max $a u; mx set $u.(2) z; nuke $u

Code:(pseudo): z=max(max(a));

fliplr

Args:(in,out): real mixed

Code:(pseudo): z=fliplr(a);

flipud

Args:(in,out): real mixed

Code:(pseudo): z=flipud(a);

floor

Args:(in,out): real mixed

Code:(pseudo): z=floor(a);

fprintf

Args:(in,out): real

Code:(mx) : mx fprintf $filename $a

Code:(pseudo): matrix row vector => line of ASCII numbers in file;

fread

Args:(in,out): real

Code:(mx) : mx fread $filename z

Code:(pseudo): row vector <= binary float numbers in file;

fscanf

Args:(in,out): real

Code:(mx) : mx fscanf $filename z

Code:(pseudo): row vector <= ASCII numbers in file;

fwrite

Args:(in,out): real

Code:(mx) : mx fwrite $filename $a

Code:(pseudo): matrix => binary float numbers in file;

ifft

Args:(in,out): real complex

Code:(mx) : set z [mx ifft "$a $a"]

Code:(pseudo): [sz sz1]=size(a); if sz*sz1~=sz, u=conj((a+im*a)');
z=conj(ifft(u)'); else z=a+im*a; end

ifft

Args:(in,out): real complex

Code:(mx) : set z [mx ifft $a]

Code:(pseudo): [sz sz1]=size(a); if sz*sz1~=sz, u=a';
z=conj(ifft(u)'); else z=a; end

imag

Args:(in) : real complex

Args:(out) : real

Code:(pseudo): z=imag(a);

join

Args:(in,out): real

Code:(mx) : mx join row [list $a $a $a] z

Code:(pseudo): z=[a;a;a];

join

Args:(in,out): real

Code:(mx) : mx join col [list $a $a $a] z

Code:(pseudo): z=[a a a];

load

Args:(in,out): real

Code:(mx) : mx load constants

Code:(pseudo): see documentation

log

Args:(in,out): real

Code:(pseudo): z=log(a);

log10

Args:(in,out): real

Code:(pseudo): z=log10(a);

max col

Args:(in,out): real

Code:(pseudo): [sz sz1]=size(a);
if sz*sz1~=sz1, [u,z]=max(a);z=z-1; else z=a*0; end

max row

Args:(in,out): real

Code:(pseudo): [sz sz1]=size(a);
if sz*sz1~=sz, [u,z]=max(conj(a'));z=z'-1; else z=a*0; end

mean col

Args:(in,out): real

Code:(pseudo): [sz sz1]=size(a);
if sz*sz1~=sz1, z=mean(a); else z=a; end

mean row

Args:(in,out): real

Code:(pseudo): [sz sz1]=size(a);
if sz*sz1~=sz, z=mean(conj(a'))'; else z=a; end

min col

Args:(in,out): real

Code:(pseudo): [sz sz1]=size(a);
if sz*sz1~=sz1, [u,z]=min(a);z=z-1; else z=a*0; end

min row

Args:(in,out): real

Code:(pseudo): [sz sz1]=size(a);
if sz*sz1~=sz, [u,z]=min(conj(a'));z=z'-1; else z=a*0; end

mul

Args:(in,out): real mixed complex

Code:(pseudo): z=a.*b;

norm

Args:(in,out): real

Code:(mx) : set z [mx norm $a -l 1]

Code:(pseudo): [sz sz1]=size(a);
if sz*sz1~=sz, z=mean(sum(abs(a')));
else z=mean(abs(a)); end;

norm

Args:(in,out): real

Code:(mx) : set z [mx norm $a -l 2]

Code:(pseudo): [sz sz1]=size(a);
if sz*sz1~=sz, z=mean(sum(abs(a').^ 2).^ 0.5);
else z=mean(abs(a)); end;

norm

Args:(in,out): real

Code:(mx) : set z [mx norm $a -l 4]

Code:(pseudo): [sz sz1]=size(a);
if sz*sz1~=sz, z=mean(sum(abs(a').^ 4).^ 0.25);
else z=mean(abs(a)); end;

norm

Args:(in,out): real

Code:(mx) : set z [mx norm $a -l inf]

Code:(pseudo): [sz sz1]=size(a);
if sz*sz1~=sz, z=mean(max(abs(a')));
else z=mean(abs(a)); end;

norm

Args:(in,out): real

Code:(mx) : set z [mx norm $a -l mininf]

Code:(pseudo): [sz sz1]=size(a);
if sz*sz1~=sz, z=mean(min(abs(a')));
else z=mean(abs(a)); end;

normalize

Args:(in,out): real

Code:(mx) : mx normalize $a z -l 1

Code:(pseudo): [sz sz1]=size(a);
if sz*sz1~=sz, z=diag(sum(abs(a')).^(-1))*a;
else z=sign(a).*a./a; end;

normalize

Args:(in,out): real

Code:(mx) : mx normalize $a z -l 2

Code:(pseudo): [sz sz1]=size(a);
if sz*sz1~=sz, z=diag((sum(abs(a').^2).^0.5).^(-1))*a;
else z=sign(a).*a./a; end;

normalize

Args:(in,out): real

Code:(mx) : mx normalize $a z -l 4

Code:(pseudo): [sz sz1]=size(a);
if sz*sz1~=sz, z=diag((sum(abs(a').^4).^0.25).^(-1))*a;
else z=sign(a).*a./a; end;

normalize

Args:(in,out): real

Code:(mx) : mx normalize $a z -l inf

Code:(pseudo): [sz sz1]=size(a);
if sz*sz1~=sz, z=diag(max(abs(a')).^(-1))*a;
else z=sign(a).*a./a; end;

normalize

Args:(in,out): real

Code:(mx) : mx normalize $a z -l mininf

Code:(pseudo): [sz sz1]=size(a);
if sz*sz1~=sz, z=diag(min(abs(a')).^ (-1))*a;
else z=sign(a).*a./a; end;

one

Args:(in,out): real mixed

Code:(pseudo): z=ones(size(a));
if max(max(abs(imag(a))))~=0, z=z+im*z; end

onediag

Args:(in,out): real

Code:(pseudo): z=a;sz=min(size(z));for i=1:sz,z(i,i)=1;end;

ones

Args:(in,out): real

Code:(mx) : mx ones 2 3 z

Code:(pseudo): z=ones(2,3);

ones

Args:(in,out): real

Code:(mx) : mx ones 1 4 z

Code:(pseudo): z=ones(1,4);

ones

Args:(in,out): real

Code:(mx) : mx ones 3 1 z

Code:(pseudo): z=ones(3,1);

p2c

Args:(in,out): real mixed

Code:(pseudo): z=$ar.*exp(im*$ai);

pabs

Args:(in) : real complex

Args:(out) : real

Code:(pseudo): z=real(a);

padd

Args:(in,out): mixed complex

Code:(pseudo): u=$ar.*exp(im*$ai);
v=$br.*exp(im*$bi); z=abs(u+v)+im*angle(u+v);

pangle

Args:(in) : real complex

Args:(out) : real

Code:(pseudo): z=imag(a);

pdiv

Args:(in,out): mixed complex

Code:(pseudo): z=$ar./$br+im*($ai-$bi);

pimag

Args:(in) : real complex

Args:(out) : real

Code:(pseudo): z=imag($ar.*exp(im*$ai));

pmul

Args:(in,out): mixed complex

Code:(pseudo): z=$ar.*$br+im*($ai+$bi);

preal

Args:(in) : real complex

Args:(out) : real

Code:(pseudo): z=real($ar.*exp(im*$ai));

prod

Args:(in,out): real mixed complex

Code:(pseudo): z=a*b;

Restrictions:: prod_size

psqr

Args:(in,out): real mixed

Code:(pseudo): z=$ar.*$ar+im*($ai+$ai);

psubtr

Args:(in,out): mixed complex

Code:(pseudo): u=$ar.*exp(im*$ai);
v=$br.*exp(im*$bi); z=abs(u-v)+im*angle(u-v);

puts

Args:(in,out): real complex

Code:(mx) : mx puts $a

Code:(pseudo): a

qsort

Args:(in,out): real

Code:(mx) : mx qsort $a -increasing z

Code:(pseudo): [sz sz1]=size(a);
if sz*sz1~=sz, z=sort(a')'; else z=a; end;

qsort

Args:(in,out): real

Code:(mx) : mx qsort $a -decreasing z

Code:(pseudo): [sz sz1]=size(a);
if sz*sz1~=sz, z=-sort(-a')'; else z=a; end;

random

Args:(in,out): real

Code:(mx) : see documentation

Code:(pseudo): see documentation

real

Args:(in) : real complex

Args:(out) : real

Code:(pseudo): z=real(a);

rem

Args:(in,out): real mixed complex

Code:(pseudo): z=a-fix(a./b).*b;

scale

Args:(in,out): real

Code:(mx) : mx scale row $a $c z

Code:(pseudo): z=diag(c)*a;

scale

Args:(in,out): real

Code:(mx) : mx scale col $a $b z

Code:(pseudo): z=a*diag(b);

set

Args:(in,out): real

Code:(mx) : mx set $a z

Code:(pseudo): z=a;

set

Args:(in,out): real

Code:(mx) : mx set "{{2 3} {4 5}}" z

Code:(pseudo): z=[2 3;4 5];

set

Args:(in,out): real

Code:(mx) : mx set "{2 3}" z

Code:(pseudo): z=[2;3];

set

Args:(in,out): real complex

Code:(mx) : set z [mx set "2 3"]

Code:(pseudo): z=2+im*3;

set

Args:(in,out): real complex

Code:(mx) : set z [mx set "{{2 3} {4 5}} {{-2 -3} {6 7}}"];
puts "$z [mx puts $z -raw]"

Code:(pseudo): z=[2 3;4 5]+im*[-2 -3;6 7];

sign

Args:(in,out): real mixed

Code:(pseudo): z=sign(a);

sin

Args:(in,out): real

Code:(pseudo): z=sin(a);

sinh

Args:(in,out): real

Code:(pseudo): z=sinh(a);

size

Args:(in) : real complex

Args:(out) : real

Code:(pseudo): z=size(a);

sqr

Args:(in,out): real mixed

Code:(pseudo): z=a.*a;

sqrt

Args:(in,out): real

Code:(pseudo): z=sqrt(a);

std col

Args:(in,out): real

Code:(pseudo): [sz sz1]=size(a);
if sz*sz1~=sz1, z=std(a); else z=0*a;end

std row

Args:(in,out): real

Code:(pseudo): [sz sz1]=size(a);
if sz*sz1~=sz, z=conj(std(conj(a'))'); else z=0*a;end

subtr

Args:(in,out): real mixed complex

Code:(pseudo): z=a-b;

sum col

Args:(in,out): real

Code:(pseudo): [sz sz1]=size(a);
if sz*sz1~=sz1, z=sum(a); else z=a; end

sum row

Args:(in,out): real

Code:(pseudo): [sz sz1]=size(a);
if sz*sz1~=sz, z=sum(conj(a'))'; else z=a; end

tan

Args:(in,out): real

Code:(pseudo): z=tan(a);

tanh

Args:(in,out): real

Code:(pseudo): z=tanh(a);

tr

Args:(in,out): real mixed

Code:(pseudo): z=a';

trace

Args:(in,out): real mixed

Code:(pseudo): [sz sz1]=size(a);
if sz~=1 & sz1~=1, z=trace(a); else z=a(1); end

unload

Args:(in,out): real

Code:(mx) : mx unload

Code:(pseudo): see documentation

unwrap

Args:(in,out): real mixed

Code:(pseudo): u=a'; z=u(:)';

value

Args:(in,out): real

Code:(mx) : mx value $a i j

Code:(pseudo): a[i,j]

weigh

Args:(in,out): real

Code:(mx) : mx weigh $a -exp 1 z

Code:(pseudo): [sz1 sz]=size(a); z=a*diag([1:sz].^(1));

zero

Args:(in,out): real mixed

Code:(pseudo): z=zeros(size(a));

zerodiag

Args:(in,out): real

Code:(pseudo): z=a;sz=min(size(z));for i=1:sz,z(i,i)=0;end;

zeromean

Args:(in,out): real mixed

Code:(pseudo): [sz sz1]=size(a);
if sz*sz1~=sz1, z=a-ones(sz,1)*mean(a); else z=a*0; end;

zeros

Args:(in,out): real

Code:(mx) : mx zeros 2 3 z

Code:(pseudo): z=zeros(2,3);

zeros

Args:(in,out): real

Code:(mx) : mx zeros 1 4 z

Code:(pseudo): z=zeros(1,4);

zeros

Args:(in,out): real

Code:(mx) : mx zeros 3 1 z

Code:(pseudo): z=zeros(3,1);

Bibliography

[1] J. K. Ousterhout, *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[2] S. van Vuuren and H. Hermansky, "Data-driven design of RASTA-like filters," in *Proc. EUROSPEECH'97*, (Rodos, Greece), pp. 409–412, 1997.

[3] S. van Vuuren and H. Hermansky, "Oregon Graduate Institute NIST speaker recognition evaluation," in *Proceedings of the NIST Speaker Recognition Workshop*, (Baltimore), 1997.

[4] S. van Vuuren and H. Hermansky, "!MESS: A modular, efficient speaker verification system," in *Proc. of Workshop on Speaker Recognition and its Commercial and Forensic Applications (RLA2C)*, (Avignon, France), April, 1998. To appear.

[5] J. Schalkwyk, J. de Villiers, S. van Vuuren, and P. Vermeulen, "CSLUsh: an extendible research environment," in *Proc. EUROSPEECH'97*, (Rodos, Greece), pp. 698–701, 1997.

[6] J. Schalkwyk and M. Fanty, "The CSLUsh toolkit for automatic speech recognition," Tech. Rep. CSLU-011-06, Center for Spoken Language Understanding, Oregon Graduate Institute, Portland, OR, 1996.

[7] J. Schalkwyk and M. Fanty, "The CSLU-c toolkit for automatic speech recognition," Tech. Rep. CSLU-012-06, Center for Spoken Language Understanding, Oregon Graduate Institute, Portland, OR, 1996.

[8] F. Bimbot and L. Mathan, "Text-free speaker recognition using an arithmetic-harmonic sphericity measure," in *Eurospeech*, (Berlin), pp. 169–172, 1993.