Tela Quick Reference

Tela Version 1.32

Starting Tela

`tela`	start interactive Tela session
`tela` *file*	run Tela on commands in *file*
`tela --help`	describe command line options

Stopping Tela

`quit()` or `exit()`	quit Tela
`TERMINATE`	(*e.g.* `C-d`) quit Tela
`INTERRUPT`	(*e.g.* `C-c`) terminate current command and return to top-level prompt

Getting Help

`help`	help on help
`help` *command* or `?`*command*	describe *command*

Command-Line Cursor Motion

`C-b`	move back one character
`C-f`	move forward one character
`C-a`	move the the start of the line
`C-e`	move to the end of the line
`M-f`	move forward a word
`M-b`	move backward a word
`C-l`	clear screen, reprinting current line at top

Inserting or Changing Text

`M-TAB`	insert a tab character
`DEL`	delete character to the left of the cursor
`C-d`	delete character under the cursor
`C-v`	add the next character verbatim
`C-t`	transpose characters at the point
`M-t`	transpose words at the point
`M-u`	uppercase the current word
`M-l`	lowercase the current word
`M-c`	capitalize the current word

Killing and Yanking

`C-k`	kill to the end of the line
`C-y`	yank the most recently killed text
`M-d`	kill to the end of the current word
`M-DEL`	kill the word behind the cursor
`M-y`	rotate the kill ring and yank the new top

Command Completion and History

`TAB`	complete a command, file, or variable name
`M-?`	list possible completions
`RET`	enter the current line
`C-p`	move 'up' through the history list
`C-n`	move 'down' through the history list
`M-<`	move to the first line in the history
`M->`	move to the last line in the history
`C-r`	search backward in the history list
`C-s`	search forward in the history list

[ ] surround optional arguments ... show one or more arguments

Shell Commands

`cd(`*dir*`)`	change working directory to *dir*
`getenv(`*envvar*`)`	return value of named environment variable
`putenv(`*name*`,` *val*`)`	set environment variable
pid `= getpid()`	get id of Tela process
out `= run(`*cmd*`,` *in*`)`	run command *cmd* as filter on *in*
rv `= system(`*cmd*`)`	execute arbitrary shell command string; retrieve return value *rv*
`!`*cmd*	execute arbitrary shell command string

Tensors

Parenthesis, preceded by a sharp-sign delimit literal tensors. Commas separate elements along the same dimension. Semicolons construct higher order tensors out of lower order ones. Elements of a tensor, in this notation, may be arbitrary expressions, provided that all dimensions agree.

`#(`*x1*`,` *x2*`, ...)`
enter a vector

`#(`*x11*`,` *x12*`;` *x21*`,` *x22*`)`
enter a 2×2 matrix

`#(#(`*x111*`,` *x112*`;` *x121*`,` *x122*`); #(`*x211*`,` *x212*`;` *x221*`,` *x222*`))`
enter a $2 \times 2 \times 2$ tensor

Ranges

base `:` *incr* `:` *limit*

Specify a range (monotonic vector) of values beginning with *base* with no elements larger than *limit.* If it is omitted, the default value of *incr* is 1.

Strings and Common Escape Sequences

A *string constant* consists of a sequence of characters enclosed double-quote marks. Single characters are enclosed in single-quote marks.

`\\`	a literal backslash
`\"`	a literal double-quote character
`\n`	newline, ASCII code 10
`\t`	horizontal tab, ASCII code 9

Index Expressions

The lowest index always is 1. Square brackets denote the tensor product of the index expressions. Surrounding square brackets by angle brackets applies mapped indexing.

var`[`*idx*`]`	select elements of a vector
var`[`*idx1*`,` *idx2*`]`	select elements of a matrix
var`[`*idx1*`,` *idx2*`, ...,` *idxN*`]`	select elements of a tensor
var`<[`*idx*`]>`	select elements of a vector
var`<[`*idx1*`,` *idx2*`]>`	select elements of a matrix
var`<[`*idx1*`,` *idx2*`, ...,` *idxN*`]>`	select elements of a tensor

idx is one of the following:

scalar	select row (column) corresponding to *scalar*
vector	select rows (columns) corresponding to the elements of *vector*
range	select rows (columns) corresponding to the elements of *range*
`:`	select all rows (columns)

Modules

`package` *mod_name*	start a new package
`global(`*var1, ...*`)`	mark variables for export
`source(`*t_file*`)`	load module or Tela file

Complex Numbers

Literal complex numbers are entered by appending a lowercase "i" or "j" to a real number.

`Re(`*z*`)`	real part
`Im(`*z*`)`	imaginary part
`arg(`*z*`)`	argument, $-\pi \le arg(z) \le \pi$
`conj(`*z*`)`	conjugate complex

Selected Built-in Variables

`Inf`	IEEE infinity
`eps`	machine precision
`pi`	π

Arithmetic and Increment Operators

x `+` *y*	addition
x `-` *y*	subtraction
x `*` *y*	element by element multiplication
x `**` *y*	matrix product
x `/` *y*	element by element right division
x `mod` *y*	remainder of division
x `^` *y*	element by element power operator
`+`*x*	unary plus (a no-operation)
`-`*x*	unary minus
x`.'`	transpose
x`'`	Hermitian transpose

Assignment Expressions

var `=` *expr*	assign expression to variable
var`[`*idx*`] =` *expr*	assign expression to indexed variable
var`<[`*idx*`]> =` *expr*	assign expression to mapped-indexed variable
x`++` (*x*`--`)	increment (decrement) *x*, return *old* value

Comparison and Boolean Operators

These operators work on an element-by-element basis. Both arguments are always evaluated.

x `<` *y*	true if *x* is less than *y*
x `<=` *y*	true if *x* is less than or equal to *y*
x `==` *y*	true if *x* is equal to *y*
x `>=` *y*	true if *x* is greater than or equal to *y*
x `>` *y*	true if *x* is greater than *y*
x `!=` *y*	true if *x* is not equal to *y*
x `&&` *y*	true if both *x* and *y* are true
x `||` *y*	true if at least one of *x* or *y* is true
`!`*bool*	true if *bool* is false

Operator Precedence

Here is a table of the operators in Tela, in order of increasing precedence.

`;`	dimension separator in array construction, statement separator
`,`	element separator in array construction
`=`	assignment, groups left to right
`:`	range generation, e.g. `1:10`, `-5:2.3:7.1`
`||`	logical "or"
`&&`	logical "and"
`== !=`	equality and inequality
`< <= >= >`	relational operators
`+ -`	addition and subtraction
`* / mod **`	multiplication, division, modulus, and matrix multiplication
`+ -`	unary plus and minus
`^`	exponentiation
`.' '`	transpose and Hermitian transpose
`!`	logical "not"

Note that "`++`" and "`--`" are statements, not operators.

Blocks and Statements

`{` [*statement*] [; *statement*] `}`
Statement block; valid wherever a single statement is valid

`for(`[*init-stmt*]`;` *cont-cond-expr*`;` [*upd-stmt*]`)` [*body-stmt*]
Like C's `for`, except that *init-stmt* and *upd-stmt* are statements, not expressions

`repeat` *statement-sequence* `until` *expr*
Execute *statement-sequence* until *expr* evaluates to non-zero

`while` (*expr*) *statement*
Test *expr*, if it is non-zero, execute *statement*. Repeat until *expr* becomes zero.

`if` (*expr*) *if-stmt* [`else` *else-stmt*]
Execute *if-stmt* if *expr* is non-zero, otherwise execute *else-stmt*.

`return`	return to calling function
`break`	exit innermost loop
`continue`	go to beginning of innermost loop

Defining Functions

`function` [*ret-list* =] *function-name* ([*arg-list*])
[*inport-export-declaration*]
`{` *function-body* `}`

ret-list may be a single identifier or a comma-separated list of identifiers delimited by square-brackets. *arg-list* is a comma-separated list of identifiers and may be empty. If a semi-colon is substituted for a comma in either *ret-list* or *arg-list* all arguments before the semi-colons are mandatory, arguments after them are optional. By default all input arguments are mandatory and all output arguments are optional.
inport-export-declaration is `local(`*var1, ...*`)`, `global(`*var1, ...*`)`, or just `global`.

Basic Matrix Generation

All matrix generating function take either all dimensions as separate arguments (e.g. `zeros(2, 3, 2)`), or as a single vector argument (e.g. `zeros(#(2 , 3, 2))`).

diag (*v* [, *n*])	create diagonal matrix out of vector, or extract diagonal of matrix. *n* selects the *n*-th super- or sub-diagonal.
eye(*n*, *m*)	create *n* by *m* identity matrix
ones(*n*, ...)	create *n* by ... tensor of ones
zeros(*n*, ...)	create *n* by ... tensor of real zeros
rzeros(*n*, ...)	
izeros(*n*, ...)	create *n* by ... tensor of integer zeros
czeros(*n*, ...)	create *n* by ... tensor of complex zeros
rand(*n*, ...)	create *n* by ... tensor of random values

Basic Matrix Manipulations

all(*a*)	check if all elements of *a* non-zero
any(*a*)	check if any elements of *a* non-zero
find(*a*)	return indices of non-zero elements
map(*f*, *a*, *d*)	map function *f* along dimension *d* of matrix *a*
mapmax(*a* [, *d*])	find maximum of *a* along dimension *d*
mapmin(*a* [, *d*])	find minimum of *a* along dimension *d*
[*b* [, *p*]] = sort(*a* [, *f*])	order elements of *a* according to order function *f*
sum(*a* [, *d*])	sum elements of *a*. If *d* is given, sum along dimension *d*
prod(*a* [, *d*])	multiply elements of *a*. If *d* is given, multiply along dimension *d*
cumsum(*a*)	array of partial sums
cumprod(*a*)	array of partial products
min(*args*)	find minimum values
max(*args*)	find maximum values
reshape(*a*, *m*, *n*, ...)	reformat *a* to be an *m* by *n* by ... tensor
flatten(*a*)	reformat *a* to a vector; *a* = `flatten()` flattens *a* in place
flip(*a* [, *d*])	reverse *a* along dimension *d* (= 1)
transpose(*a* [, *perm*])	real transposition; indices permuted according to *perm*
herm(*a* [, *perm*])	Hermitian conjugation; indices permuted according to *perm*

Arithmetic Functions

abs(*a*)	absolute value
abs2(*a*)	square of absolute value
sign(*a*)	sign function
HeavisideTheta(*a*)	unit step function
floor(*a*)	largest integer which is smaller than *a*
ceil(*a*)	smallest integer which is larger than *a*
round(*a*)	round to nearest integer

Transcendental Functions

sqrt(*a*)	square root
exp(*a*) log(*a*)	exponential and logarithm
sin(*a*) cos(*a*)	trigonometric functions
tan(*a*) cot(*a*)	
sec(*a*) csc(*a*)	
asin(*a*) acos(*a*)	inverse trigonometric functions
atan(*a*)	
sinh(*a*) cosh(*a*)	hyperbolic trig functions
tanh(*a*)	
asinh(*a*) acosh(*a*)	inverse hyperbolic trig functions
atanh(*a*)	
atan2(*x*, *y*)	four-quadrant arctangent

Size Inquiry

rank(*a*)	rank of tensor *a*
[*n* [, ...]] = size(*a*)	dimensions of tensor *a*
v = size(*a*)	dimension vector of tensor *a*
length(*a*)	total number of elements in *a*

Linear Algebra

y = axpy(α, *x*)	$y = y + \alpha x$ for scalar α and arrays *x* and *y*
matprod(*a*, *b* [, *fa* [, *fb*]])	matrix product, *fa* or *fb* are 'n': no operation, 't': transpose, or 'h': Hermitian conjugate
x = linsolve(*a*, *b*)	solve linear equations $ax = b$
[*l*, *u*, *p*] = LU(*a*)	computes the LU factorization
[*lu* [, *p*]] = LU(*a*)	LU factorization, *l*, *u* packed into one matrix *lu*
x = LUbacksubst(*lu*, *p*, *b*)	solve linear equations $p\, lu\, x = b$
chol(*a*)	Cholesky factorization
det(*a*)	compute determinant
[*d* [, *v*]] = eig(*a*)	eigenvalues *d* and right eigenvectors *v*
inv(*a*)	invert square matrix
[u, s, v] = SVD(*a*)	singular value decomposition
s = SVD(*a*)	singular values

Signal Processing

FFT(*u* [, *d*])	fast Fourier transform along dimension *d*; default: $d = 1$
invFFT(*u*)	inverse fast Fourier transform
realFFT(*u* [, *d*])	fast Fourier transform of real *u* along dimension *d*; default: $d = 1$
invrealFFT(*u* [, *d* [, *symm*]])	inverse real fast Fourier transform along dimension *d*; default: $d = 1$. *symm* is `"even"` or `"odd"`
sinFFT(*u* [, *d*])	sine fast Fourier transform
invsinFFT(*u*)	inverse sine fast Fourier transform
cosFFT(*u* [, *d*])	cosine fast Fourier transform
invcosFFT(*u*)	inverse cosine fast Fourier transform
sinqFFT(*u* [, *d*])	quarter-wave sine fast Fourier transform
invsinqFFT(*u*)	inverse quarter-wave sine fast Fourier transform
cosqFFT(*u* [, *d*])	quarter-wave cosine fast Fourier transform
invcosqFFT(*u*)	inverse quarter-wave cosine fast Fourier transform

Strings

`strmat(`*s1*`,` *s2*`, ...)`	create matrix of strings
`strmat2(`*s* [, *sep*]`)`	create matrix from string *s* with newlines or *sep* separating rows
`str2num(`*s*`)`	convert string to number
`tostring(`*x*`)`	convert integer vector *x* to string
`streq(`*s1*`,` *s2*`)`	compare strings (full length)
`strstarteq(`*s1*`,` *s2*`)`	compare strings (active length)
`strstr(`*haystack*`,` *needle*`)`	search *needle* in *haystack*

C-style Input and Output

`fopen(`*name*`,` *mode*`)`	open file *name*; *mode* is "`r`", "`w`", "`a`"
`fclose(`*file*`)`	close *file*
`printf(`*fmt*`, ...)`	formatted output to `stdout`
`fprintf(`*file*`,` *fmt*`, ...)`	formatted output to *file*
`sprintf(`*fmt*`, ...)`	formatted output to string
`fgetc(`*file*`)`	read single character from *file*
`[`*s* [, *tc*]`] = fgets(`*file*`)`	read from *file*, return terminal char *tc* (usually: newline)
`fread(`*file*`,` *size*`,` *prec*`)`	read binary data files
`feof(`*file*`)`	determine if pointer is at EOF
`remove(`*name*`)`	delete *name*

Other Input and Output functions

`disp(`*var*`)`	display value of *var* to screen
`format(`*fmt*`, ...)`	formatted output to stdout
`fformat(`*file*`,` *fmt*`, ...)`	formatted output to *file*
s `= sformat(`*file*`,` *fmt*`, ...)`	formatted output to string *s*
`fparse(`*file*`,` *fmt*`, ...)`	read from *file*
`save(`*file*`,` *var*`, ...)`	save variables in *file*
`load(`*file*`)`	load variables from *file*
`import(`*file* [, *suf*]`)`	import HDF or MAT file, append *suf* to each variable name
`import1(`*file* [, *label*]`)`	import HDF (SDS) tagged with *label*, plain ASCII, or D-style ASCII file
`import_CDF(`*file* [, *vname*, ...]`)`	import complete CDF file or selected *varnames*
`import_PBM(`*file*`)`	reads PBM format image file
`export_ASCII(`*file*`,` *var*`)`	write *var* to *file*
`export_PBM(`*file*`,` *var*`)`	write 8-bit, or 24-bit PBM file
`export_PBM(`*file*`,` *r*`,` *b*`,` *g*`)`	
`export_matlab(`*file* [, *vname*, ...]`)`	save all or selected variables to *file*
`export_matlab2(`*file*`,` *var*`,` *vname* [, *var*, *vname*]`)`	save selected variables to *file* while renaming them
`export_netCDF(`*file* [, *vname*, ...]`)`	save variables in netCDF format
`export_CDF(`*file* [, *flags* [, *vname*, ...]]`)`	writes named global variables to CDF file

Miscellaneous Functions

`eval(`*str*`)`	evaluate *str* as a command
x `= evalexpr(`*str*`)`	like `eval`, but return result *y*
`error(`*message*`)`	print message and return to top level
`clear(`*vname* [, *vname*]`)`	clear variables
`hide(`*vname* [, *vname*]`)`	set hidden attribute
`unhide(`*vname* [, *vname*]`)`	unset hidden attribute
`autoglobal(`*vname* [, *vname*]`)`	set autoglobal attribute
`isdefined(`*vname*`)`	check for existence of *vname*
`isundefined(`*vname*`)`	check for non-existence of *vname*
`whos(`["hidden"]`)`	list current variables
`[`*oldpath* [, *maxelems*]`] = telapath(`[*newpath*]`)`	get or set search path for Tela files

Basic Plotting (PlotMTV)

`plot(`plot-spec [, plot-spec, ...]`)`

2D plot, where plot-spec is the pair or triple *x*, *y* [, *opt*]. *x* and *y* are vectors or matrices. *opt* is a comma separated list of option strings and option arguments.

`plot3(`*x*`,` *y*`,` *z* [, *opt*]`)`

Plot parametric space curves.

`mesh(`*z* [, *opt*]`)`

Plot matrix *z* as a 3D mesh.

`mesh(`*z*`, "xgrid" ,` *xv*`, "ygrid",` *yv* [, *opt*]`)`

Plot matrix *z* in a non-uniform 3D mesh.

`contour(`*z* [, *opt*]`)`

Plot matrix *z* as a filled contour plot.

`contour(`*z*`, "xgrid" ,` *xv*`, "ygrid",` *yv* [, *opt*]`)`

Plot matrix *z* as non-uniform contour plot.

`contour3(`*z*`, "xgrid" ,` *xv*`, "ygrid" ,` *yv*`, "zgrid",` *zv* [, *opt*]`)`

Plot the 3D array as a "volume" plot.

`pcolor(`*z* [, *opt*]`)`

Plot the matrix *z* as a pseudo-color density plot.

`pcolor(`*z*`, "xgrid" ,` *xv*`, "ygrid",` *yv* [, *opt*]`)`

Plot the matrix *z* as a non-uniform pseudo-color plot.

`vplot(`*x*`,` *y*`,` *vx*`,` *vy* [, *opt*]`)`

Plot 2D vector plot of the vector field (*vy*, *vx*).

Plot Options (PlotMTV)

`annotate(`*key*`,` *val* [, ...]`)`	decorate graph
`hold(0`|`1)`	hold mode off|on
`holdmode(`[1|2|3]`)`	query/set hold-mode, 1: overlay, 2: paging, 3: stacking
`colormap(`*r*`,` *g*`,` *b*`)`	

The following *key*, *val* pairs are valid, where *str*: string, *i*: non-negative integer, *x*: real, *b*: 0 or 1, *v*: vector of reals

`toplabel` *str*	plot title
`subtitle` *str*	plot sub-title
`xlabel` *str*	x-axis label
`ylabel` *str*	y-axis label
`zlabel` *str*	z-axis label
`comment` *str*	text on top right hand side corner
`xmin  xmax`	x-axis range
`ymin  ymax`	y-axis range
`zmin  zmax` *x*	z-axis range
`xgrid  ygrid` *v*	x/y grid points
`cmin  cmax` *x*	min/max contour value
`nsteps` *i*	number of contours
`cstep` *x*	contour spacing
`conststyle` *i*	1: contours, 2: colored, 3: 3D surface
`hiddenline` *b*	wire-frame or surface

Getting System Information

`tic() toc()`	start/stop timer
`meminuse()`	size of the current Tela process in bytes
`info()`	information about object sizes and counts
`disasm(`*func*`)`	disassemble function *func*

Getting Performance Data

p `= perf()`	fetch operation counters
`cputime(`[*p*]`)`	time in seconds used by the current session or *p* (obtained with `perf()`)
`Mflops(`[p]`)`	approximate megaflop rate of current session or *p* (obtained with `perf()`)
`Mips(`[p]`)`	approximate flat-code instruction rate of current session or *p* (obtained with `perf()`)

`[`*ins*`,` *ops*`] = GetInstructionData(`*p*`,` *mnem*`)`: Fetch the number of instructions *ins* and number of operations *ops* of mnemonic operation *mnem* in the performance counter data *p*.
