Yorick Language Reference

(for version 1)

Starting and Quitting Yorick

To enter Yorick, just type its name: `yorick`

normal Yorick prompt	`>`
prompt for continued line	`cont>`
prompt for continued string	`quot>`
prompt for continued comment	`comm>`
prompt in debug mode	`dbug>`

`quit`	close all open files and exit Yorick

Getting Help

Most Yorick functions have online documentation.

`help`	help on using Yorick help
`help, f`	help on a specific function f
`info, v`	information about a variable v

Error Recovery

To abort a running Yorick program type `C-c`

To enter Yorick's debug mode after an error, type return in response to the first prompt after the error occurs.

Array Data Types

The basic data types are:

`char`	one 8-bit byte, from 0 to 255
`short`	compact integer, at least 2 bytes
`int`	**logical results**– 0 false, 1 true, at least 2 bytes
`long`	**default integer**– at least 4 bytes
`float`	at least 5 digits, $10^{\pm 38}$
`double`	**default real**– 14 or 15 digits, usually $10^{\pm 308}$
`complex`	**re** and **im** parts are double
`string`	0-terminated text string
`pointer`	pointer to an array

A compound data type *compound_type* can be built from any combination of basic or previously defined data types as follows:

```
  struct compound_type {
      type_name_A memb_name_1 ;
      type_name_B memb_name_2(dimlist) ;
      type_name_C memb_name_3,memb_name_4(dimlist) ;
      ...
  }
```

A *dimlist* is a comma delimited list of dimension lengths, or lists in the format returned by the `dimsof` function, or ranges of the form `min_index : max_index`. (By default, `min_index` is 1.)

For example, the `complex` data type is predefined as:

```
  struct complex { double re, im; }
```

Constants

By default, an integer number is a constant of type `long`, and a real number is a constant of type `double`. Constants of the types `short`, `int`, `float`, and `complex` are specified by means of the suffices `s`, `n`, `f`, and `i`, respectively. Here are some examples:

`char`	`'\0', '\1', '\x7f', '\177', 'A', '\t'`
`short`	`0s, 1S, 0x7fs, 0177s, -32766s`
`int`	`0N, 1n, 0x7Fn, 0177n, -32766n`
`long`	`0, 1, 0x7f, 0177, -32766, 1234L`
`float`	`.0f, 1.f, 1.27e2f, 0.00127f, -32.766e3f`
`double`	`0.0, 1.0, 127.0, 1.27e-3, -32.766e-33`
`complex`	`0i, 1i, 127.i, 1.27e-3i, -32.766e-33i`
`string`	`"", "Hello, world!", "\tTab\n2nd line"`

The following escape sequences are recognized in type `char` and type `string` constants:

`\n`	newline
`\t`	tab
`\"`	double quote
`\'`	single quote
`\\`	backslash
`\ooo`	octal number
`\xhh`	hexadecimal number
`\a`	alert (bell)
`\b`	backspace
`\f`	formfeed (new page)
`\r`	carriage return

Defining Variables

var `=` *expr*	redefines *var* as the value of *expr*
var `= [ ]`	undefines *var*

Any previous value or data type of *var* is forgotten. The *expr* can be a data type, function, file, or any other object.

The `=` operator is a binary operator which has the side effect of redefining its left operand. It associates to the right, so

var1 `=` *var2* `=` *var3* `=` *expr*	initializes all three *var* to *expr*

Arithmetic and Comparison Operators

From highest to lowest precedence,

`^`	raise to power
`* / %`	multiply, divide, modulo
`+ -`	add, subtract (also unary plus, minus)
`<< >>`	shift left, shift right
`>= < <= >`	(not) less, (not) greater (`int` result)
`== !=`	equal, not equal (`int` result)
`&`	bitwise and
`~`	bitwise xor (also unary bitwise complement)
`|`	bitwise or
`=`	redefine or assign

Any binary operator may be prefixed to `=` to produce an increment operator; thus `x*=5` is equivalent to `x=x*5`. Also, `++x` and `--x` are equivalent to `x+=1` and `x-=1`, respectively. Finally, `x++` and `x--` increment or decrement `x` by 1, but return the value of `x` **before** the operation.

Creating Arrays

`[` *obj1, obj2, ..., objN* `]`	build an array of N objects

The *objI* may be arrays to build multi-dimensional arrays.

`array(`*value, dimlist*`)`	add dimensions *dimlist* to *value*
`array(`*type_name, dimlist*`)`	return specified array, all zero

`span(`*start, stop, n*`)`	*n* equal stepped values from *start* to *stop*
`spanl(`*start, stop, n*`)`	*n* equal ratio values from *start* to *stop*
`grow,` *var, sfx1, sfx2, ...*	append *sfx1*, *sfx1*, etc. to *var*

These functions may be used to generate multi-dimensional arrays; use `help` for details.

Indexing Arrays

x`(`*index1, index2, ..., indexN*`)`	is a subarray of the array *x*

Each index corresponds to one dimension of the *x* array, called the **ID** in this section (the two exceptions are noted below). The *index1* varies fastest, *index2* next fastest, and so on. By default, Yorick indices are 1-origin. An *indexI* may specify multiple index values, in which case the result array will have one or more dimensions which correspond to the **ID** of *x*. Possibilities for the *indexI* are:

- scalar index
  Select one index. No result dimension will correspond to **ID**.
- nil (or omitted)
  Select the entire **ID**. One result dimension will match the **ID**.
- index range `start:stop` or `start:stop:step`
  Select `start`, `start+step`, `start+2*step`, etc. One result dimension of length `1+(stop-start)/step` and origin 1 will correspond to **ID**. The default `step` is 1; it may be negative. In particular, `::-1` reverses the order of **ID**.
- index list
  Select an arbitrary list of indices – the index list can be any array of integers. The dimensions of the index list will replace the **ID** in the result.
- pseudo-index `-`
  Insert a unit length dimension in the result which was not present in the original array *x*. There is no **ID** for a `-` index.
- rubber-index `..` or `*`
  The **ID** may be zero or more dimensions of *x*, forcing it indexN to be the final actual index of *x*. A `..` preserves the actual indices, `*` collapses them to a single index.
- range function *ifunc* or *ifunc*:range
  Apply a range function to all or a subset of the **ID**; the other dimensions are "spectators"; multiple *ifunc* are performed successively from left to right.

Function results and expressions may be indexed directly, e.g.:
f`(`*a,b,c*`)(`*index1,index2*`)` or `(2*`*x*`+1)(`*index1,index2,index3*`)`

If the left hand operand of the `=` operator is an indexed array, the right hand side is converted to the type of the left, and the specified array elements are replaced. Do not confuse this with the redefinition operation *var*`=`:

x`(`*index1, index2, ..., indexN*`)=` *expr*	assign to a subarray of *x*

Array Conformability Rules

Operands may be arrays, in which case the operation is performed on each element of the array(s) to produce an array result. Binary operands need not have identical dimensions, but their dimensions must be *conformable*. Two arrays are conformable if their first dimensions *match*, their second dimensions *match*, their third dimensions *match*, and so on up to the number of dimensions in the array with the fewer dimensions. Two array dimensions *match* if either of the following conditions is met:

- the dimensions have the same length
- one of the dimensions has unit length (1 element)

Unit length or missing dimensions are broadcast (by copying the single value) to the length of the corresponding dimension of the other operand. The result of the operation has the number of dimensions of the higher rank operand, and the length of each dimension is the longer of the lengths in the two operands.

Logical Operators

Yorick supports C-style logical AND and OR operators. Unlike the arithmetic and comparison operators, these take only scalar operands and return a scalar `int` result. Their precedence is between `|` and `=`.

The right operand is not evaluated at all if the value of the left operand decides the result value; hence the left operand may be used to determine whether the evaluation of the right operand would lead to an error.

`&&`	logical and (scalar `int` result)
`||`	logical or (scalar `int` result)

The logical NOT operator takes an array or a scalar operand, returning `int` 1 if the operand was zero, 0 otherwise. Its precedence is above `^`.

`!`	logical not (`int` result)

The ternary operator selects one of two values based on the value of a scalar *condition*:

condition `?` *true_expr* `:` *false_expr*

Its precedence is low, and it must be parenthesized in a function argument list or an array index list to prevent confusion with the `:` in the index range syntax. Like `&&` and `||`, the expression which is rejected is not evaluated at all.

Calling Functions

f`(`*arg1, ..., argN*`)`	invoke *f* as a function
f`,` *arg1, ..., argN*	invoke *f* as a subroutine, discard return

Arguments which are omitted are passed to the function as nil. In addition to positional arguments, a function (invoked by either of the above two mechanisms). Keyword arguments look like this:

f`,` *arg1, keyA= exprA, keyB= exprB, arg2, ...*

where *keyA* and *keyB* are the names of keyword arguments of the function *f*. Omitted keywords are passed to *f* as nil values. Keywords typically set optional values which have defaults.

Defining Functions

A function of N dummy arguments is defined by:

```
func func_name( dummy1, dummy2, ..., dummyN)
{
    body_statements
}
```

If the function has no dummy arguments, the first line of the definition should read:

```
func func_name
```

Mark output parameters with a `&`, as *dummy2* here:

```
func func_name( dummy1, &dummy2, dummy3)
```

If the function will take keyword arguments, they must be listed after all positional arguments and marked by a `=`:

```
func func_name( ..., dummyN, key1=, ..., keyN=)
```

If the function allows an indeterminate number of positional arguments (beyond those which can be named), place the special symbol `..` after the final dummy argument, but before the first keyword. For example, to define a function which takes one positional argument, followed by an indeterminate number of positional arguments, and one keyword, use:

```
func func_name(dummy1, .., key1=)
```

The function `more_args()` returns the number of unread actual arguments corresponding to the `..` indeterminate dummy argument. The function `next_arg()` reads and returns the next unread actual argument, or nil if all have been read.

Variable Scope

`local` *var1, var2, ..., varN*	give the *varI* local scope
`extern` *var1, var2, ..., varN*	give the *varI* external scope

If a variable *var* has local scope within a function, any value associated with *var* is temporarily replaced by nil on entry to the function. On return from the function, the external value of *var* is restored, and the local value is discarded.

If a variable *var* has external scope within a function, references to *var* within the function refer to the *var* in the "nearest" calling function for which *var* has local scope (that is, to the most recently created *var*).

The `*main*` function has no variables of local scope; all variables created at this outermost level persist until they are explicitly undefined or redefined.

Dummy or keyword arguments always have local scope.

In the absence of a `extern` or `local` declaration, a variable *var* has local scope if, and only if, its first use within the function is as the left operand of a redefinition, *var*`=` *expr*.

Returning from Functions

`return` *expr*	return *expr* from current function

The *expr* may be omitted to return nil, which is the default return value if no `return` statement is encountered.

`exit,` *msg*	return from all functions, printing *msg*
`error,` *msg*	halt with error, printing *msg*

Compound Statements

Yorick statements end with a `;` or end-of-line if the resulting statement would make sense.

Several Yorick statements can be combined into a single compound statement by enclosing them in curly braces:

```
{
    statement1
    statement2
   ...
}
```

The bodies of most loops and `if` statements are compound.

Conditional Execution

A Yorick statement can be executed or not based on the value of a scalar *condition* (0 means don't execute, non-0 means execute):

```
if ( condition)  statementT
```

or, more generally,

```
if ( condition)  statementT
else  statementF
```

Several `if` statements may be chained as follows:

```
if ( condition1)  statement1
else if ( condition2)  statement2
else if ( condition3)  statement3
...
else  statementF
```

Loops

Yorick has three types of loops:

```
while ( condition)  body_statement
do  body_statement while ( condition)
for ( init_expr ; test_expr ; inc_expr)  body_statement
```

The *init_expr* and *inc_expr* of a `for` loop may be comma delimited lists of expressions. They or the *test_expr* may be omitted. In particular, `for (;;) ...` means "do forever". If there is a *test_expr*, the *body_statement* of the `for` loop will execute until it becomes false (possibly never executing). After each pass, but before the *test_expr*, the *inc_expr* executes. A `for` loop to make N passes through its *body_statement* might look like this:

```
for (i=1 ; i<=N ; i++) body_statement
```

Within a loop body, the following statements are legal:

`break`	exit the current loop now
`continue`	abort the current pass through the current loop

For more complex flow control, Yorick supports a `goto`:

`goto` *label*	go to the statement after *label*
label`:` *statement*	mark *statement* as a `goto` target

Yorick Function Reference

(for version 1)

Including Source Files

`#include "`*filename.i*`"`	insert contents of *filename*

This is a parser directive, NOT an executable statement. Yorick also provides two forms of executable include statements:

`include, "`*filename.i*`"`	parse contents of *filename.i*
`require, "`*filename.i*`"`	parse *filename.i* if not yet parsed

The effect of the `include` function is not quite immediate, since any tasks (`*main*` programs) generated cannot execute until the task which called `include` finishes.

The `require` function should be placed at the top of a file which represents a package of Yorick routines that depends on functions or variables defined in another package *filename.i*.

The *filename.i* ends with a `.i` suffix by convention.

Comments

```
  /* Yorick comments begin with slash-asterisk,
     and end with asterisk-slash.  A comment
     of any size is treated as a single blank.  */
```

Since `/* ... */` comments do not nest properly, Yorick supports C++ style comments as well:

```
   statement   // remainder of line is comment (C++)
   // Prefix a double slash to each line to comment out
   // a block of lines, which may contain comments.
```

Issuing Shell Commands

You can execute a system command, returning to Yorick when the command completes, by prefixing the command line with `$`:

```
  $any shell command line
```

This is a shorthand for the `system` function:

`system,` *shell_string*	pass *shell_string* to a system shell

You need to use the `system` function if you want to compute the *shell_string*; otherwise `$` is more convenient.

Note that the cd (change directory) shell command and its relatives will not have any effect on Yorick's working directory. Instead, use Yorick's `cd` function to change it's working directory:

`cd,` *path_name*	change Yorick's default directory
`get_cwd()`	return Yorick's current working directory

The following functions also relate to the operating system:

`get_home()`	return your home directory
`get_env(`*env_string*`)`	return environment variable *env_string*
`get_argv()`	return the command line arguments

Matrix Multiplication

The `*` binary operator normally represents the product of its operands element-by-element, following the same conformability rules as the other binary operators. However, by marking one dimension of its left operand and one dimension of its right operand with `+`, `*` will be interpreted as a matrix multiply along the marked dimensions. The marked dimensions must have the same length. The result will have the unmarked dimensions of the left operand, followed by the unmarked dimensions of the right operand.

For example, if `x` is a 12-by-25-by-35 array, `y` and `z` are vectors of length 35, and `w` is a 9-by-12-by-7 array, then:

`x(,,+)*y(+)`	is a 12-by-25 array
`y(+)*z(+)`	is the inner product of `y` and `z`
`x(+,,)*w(,+,)`	is a 25-by-35-by-9-by-7 array

Using Pointers

A scalar of type `pointer` points to a Yorick array of any data type or dimensions. Unary `&` returns a `pointer` to its argument, which can be any array valued expression. Unary `*` dereferences its argument, which must be a scalar of type `pointer`, returning the original array. A dereferenced pointer may itself be an array of type `pointer`. The unary `&` and `*` bind more tightly than any other Yorick operator except `.` and `->` (the member extraction operators), and array indexing *x*`(..)`:

`&`*expr*	return a scalar `pointer` to *expr*
`*`*expr*	dereference *expr*, a scalar pointer

Since a `pointer` always points to a Yorick array, Yorick can handle all necessary memory management. Dereference `*` or `->`, copy by assignment `=`, or compare to another pointer with `==` or `!=` are the only legal operations on a `pointer`. A pointer to a temporary *expr* makes sense and may be useful.

The purpose of the `pointer` data type is to deal with several related objects of different types or shapes, where the type or shape changes, making `struct` inapplicable.

Instancing Data Structures

Any data type *type_name* — basic or defined by `struct` — serves as a type converter to that data type. A nil argument is converted to a scalar zero of the specified type. Keywords matching the member names can be used to assign non-zero values to individual members:

type_name`()`	scalar instance of *type_name*, zero value
type_name`(`*memb_name_1*`=`*expr_1*`,...)`	scalar *type_name*

The `.` operator extracts a member of a data structure. The `->` operator dereferences a pointer to the data structure before extracting the member. For example:

```
struct Mesh { pointer x, y; long imax, jmax; }
mesh= Mesh(x=&xm,
           imax=dimsof(xm)(1), jmax=dimsof(xm)(2));
mesh.y= &ym;        mptr= &mesh;
print, mesh.x(2,1:10), mptr->y(2,1:10);
```

Index Range Functions

Range functions are executed from left to right if more than one appears in a single index list. The following range functions reduce the rank of the result, like a scalar index:

`min`	minimum of values along index
`max`	maximum of values along index
`sum`	sum of values along index
`avg`	average of values along index
`rms`	root mean square of values along index
`ptp`	peak-to-peak of values along index
`mnx`	index at which minimum occurs
`mxx`	index at which maximum occurs

The following functions do not change the rank of the result, like an index range. However, the length of the index is changed as indicated by +1, -1, or 0 (no change):

`cum, psum`	+1, 0, partial sums of values along index
`dif`	-1, pairwise differences of adjacent values
`zcen`	-1, pairwise averages of adjacent values
`pcen`	+1, pairwise averages of adjacent interior values
`uncp`	-1, inverse of pcen (point center) operation

For example, given a two-dimensional array `x`, `x(min, max)` returns the largest of the smallest elements along the first dimension. To get the smallest of the largest elements along the second dimension, use `x(, max)(min)`.

Elementary Functions

`abs, sign`	absolute value, arithmetic sign
`sqrt`	square root
`floor, ceil`	round down, round up to integer
`conj`	complex conjugation
`pi`	the constant 3.14159265358979323846...
`sin, cos, tan`	trigonometric functions (of radians)
`asin, acos, atan`	inverse trigonometric functions
`sinh, cosh, tanh, sech, csch`	hyperbolic functions
`asinh, acosh, atanh`	inverse hyperbolic functions
`exp, log, log10`	exponential and logarithmic functions
`min, max`	find minimum, maximum of array
`sum, avg`	find sum, average of array
`random`	random number generator

The `atan` function takes one or two arguments; `atan(t)` returns a value in the range $(-\pi/2, \pi/2])$, while `atan(y,x)` returns the counterclockwise angle from $(1,0)$ to (x,y) in the range $(-\pi,\pi])$.

The `abs` function allows any number of arguments; for example, `abs(x, y, z)` is the same as `sqrt(x^2 + y^2 + z^2)`. The `sign` satisfies `sign(0)==1` and `abs(z)*sign(z)==z` always (even when `z` is `complex`).

The `min` and `max` functions return a scalar result when presented with a single argument, but the pointwise minimum or maximum when presented with multiple arguments.

The `min`, `max`, `sum`, and single argument `abs` functions return integer results when presented integer arguments; the other functions will promote their arguments to a real type and return reals.

Information About Variables

`print`, *var1, var2, ...*	print the values of the *varI*
`info`, *var*	print a description of *var*
`dimsof(`*x*`)`	returns [# dimensions, length1, length2, ...]
`orgsof(`*x*`)`	returns [# dimensions, origin1, origin2, ...]
`numberof(`*x*`)`	returns number of elements (product of `dimsof`)
`typeof(`*x*`)`	returns name of data type of x
`structof(`*x*`)`	returns data type of x
`is_array(`*x*`)`	returns 1 if x is an array, else 0
`is_func(`*x*`)`	returns 1 or 2 if x is an function, else 0
`is_void(`*x*`)`	returns 1 if x is nil, else 0
`is_range(`*x*`)`	returns 1 if x is an index range, else 0
`is_stream(`*x*`)`	returns 1 if x is a binary file, else 0
`am_subroutine()`	1 if current function invoked as subroutine

The `print` function returns a string array of one string per line if it is invoked as a function. Using `print` on files, bookmarks, and other objects usually produces some sort of useful description. Also, `print` is the default function, so that

expr

is equivalent to `print,` *expr* (if *expr* is not a function).

Reshaping Arrays

`reshape,` *x, type_name, dimlist*	masks shape of x

Don't try to use this unless (1) you're an expert, and (2) you're desperate. It is intended mainly for recovering from misfeatures of other programs, although there are a few legitimate uses within Yorick.

Logical Functions

`allof(`*x*`)`	returns 1 if every element of x is non-zero
`anyof(`*x*`)`	returns 1 if any element of x is non-zero
`noneof(`*x*`)`	returns 1 if no element of x is non-zero
`nallof(`*x*`)`	returns 1 if any element of x is zero
`where(`*x*`)`	returns list of indices where x is non-zero
`where2(`*x*`)`	human-readable variant of `where`

Interpolation and Lookup Functions

In the following function, y and x are one-dimensional arrays which determine a piecewise linear function $y(x)$. The x must be monotonic. The xp (for x-prime) can be an array of any dimensionality; the dimensions of the result will be the same as the dimensions of xp.

`digitize(`*xp, x*`)`	returns indices of xp values in x
`interp(`*y, x, xp*`)`	returns yp, xp interpolated into $y(x)$
`integ(`*y, x, xp*`)`	returns the integrals of $y(x)$ from $x(1)$ to xp

Note that `integ` is really an area-conserving interpolator. If the xp coincide with x, you probably want to use

```
(y(zcen)*x(dif))(cum)
```

instead.

The on-line `help` documentation for `interp` describes how to use `interp` and `integ` with multidimensional y arrays.

Sorting

`sort(`*x*`)`	return index list which sorts x

That is, *x*`(sort(`*x*`))` will be in non-decreasing order (x can be an integer, real, or string array). The on-line `help` documentation for `sort` explains how to sort multidimensional arrays.

`median(`*x*`)`	return the median of the x array

Consult the on-line `help` documentation for `median` for use with multidimensional arrays.

Transposing

`transpose(`*x*`)`	transpose the 2-D array x
`transpose(`*x, permutation*`)`	general transpose

The *permutation* is a comma delimited list of cyclic permutations to be applied to the indices of x. Each cyclic permutation may be:

- a list of dimension numbers [*n1, n2, ..., nN*] to move dimension number *n1* (the first dimension is number 1, the second number 2, and so on) to dimension number *n2*, *n2* to *n3*, and so on, until finally *nN* is moved to *n1*.
- a scalar integer n to move dimension number 1 to dimension number n, 2 to n+1, and so on, cyclically permuting all of the indices of x.

In either case, n or *nI* can be non-positive to refer to indices relative to the **final** dimension of x. That is, 0 refers to the final dimension of x, -1 to the next to last dimension, and so on. Thus,

```
transpose(x, [1,0])
```

swaps the first and last dimensions of x.

Manipulating Strings

Yorick type `string` is a pointer to a 0-terminated array of `char`. A string with zero characters – `""` – differs from a zero pointer – `string(0)`. A `string` variable s can be converted to a `pointer` to a 1-D array of `char`, and such a `pointer` p can be converted back to a `string`:

```
p= pointer( s);
s= string( p);
```

These conversions copy the characters, so you can't use the pointer p to alter the characters of s.

Given a string or an array of strings s:

`strlen(`*s*`)`	number of characters in each element of s
`strmatch(`*s, pat*`)`	1 if *pat* occurs in s
`strmatch(`*s, pat*`, 1)`	1 if *pat* occurs in s, case insensitive
`strpart(`*s, m:n*`)`	returns substring of s
`strtok(`*s, delims*`)`	gets first token of s
`strtok(`*s*`)`	gets first whitespace delimited token of s

The `strtok` function returns a 2-by-`dimsof(`*s*`)` array of strings — the first token followed by the remainder of the string. The token will be `string(0)` if no tokens were present; the remainder of string will be `string(0)` if there are no characters after the token.

Advanced Array Indexing

- A scalar index or the `start` and `stop` of an index range may be non-positive to reference the elements near the end of a dimension. Hence, 0 refers to the final element, -1 refers to the next to last element, -2 to the element before that, and so on. For example, *x*`(2:-1)` refers to all but the first and last elements of the 1-D array x. This convention does **NOT** work for an index list.
- A range function *ifunc* may be followed by a colon and an index range `start:stop` or `start:stop:step` in order to restrict the indices to which the range function applies to a subset of the entire dimension. Hence, *x*`(min:2:-1)` returns the minimum of all the elements of the 1-D array x, excluding the first and last elements.
- An index specified as a scalar, the `start` or `stop` of an index range, or an element of an index list may exceed the length of the indexed dimension **ID**, provided that the entire indexing operation does not overreach the bounds of the array. Thus, if y is a 5-by-6 array, then *y*`(22)` refers to the same datum as *y*`(2,5)`.
- The expression *z*`(..)` — using the rubber-index operator `..` — refers to the entire array z. This is occasionally useful as the left hand side of an assignment statement in order to force broadcasting and type conversion of the right hand expression to the preallocated type and shape z.
- The expression *z*`(*)` — using the rubber-index operator `*` — collapses a multidimensional array z into a one-dimensional array. Even more useful as *z*`(*,)` to preserve the final index of an array and force a two-dimensional result.

Generating Simple Meshes

Many Yorick calculations begin by defining an array of x values which will be used as the argument to functions of a single variable. The easiest way to do this is with the `span` or `spanl` function:

```
x= span(x_min, x_max, 200);
```

This gives 200 points equally spaced from `x_min` to `x_max`.

A two dimensional rectangular grid is most easily obtained as follows:

```
x= span(x_min, x_max, 50)(, -:1:40);
y= span(y_min, y_max, 40)(-:1:50, );
```

This gives a 50-by-40 rectangular grid with `x` varying fastest. Such a grid is appropriate for exploring the behavior of a function of two variables. Higher dimensional meshes can be built in this way, too.

Yorick I/O Reference

(for version 1)

Opening and Closing Text Files

f= `open`(*filename, mode*)	open *filename* in *mode*
`close,` *f*	close file *f* (automatic if *f* redefined)

The *mode* is a string which announces the type of operations you intend to perform: `"r"` (the default if *mode* is omitted) means read operations only, `"w"` means write only, and destroy any existing file *filename*, `"r+"` means read/write, leaving any existing file *filename* intact. Other *mode* values are also meaningful; see `help`.

The file variable *f* is a distinct data type in Yorick; text files have a different data type than binary files. The `print` or `info` function will describe the file. The `close` function is called implicitly when the last reference to a file disappears.

Reading Text

`read,` *f, var1, var2, ..., varN*	reads the *varI* from file *f*
`read,` *var1, var2, ..., varN*	reads the *varI* from keyboard
`read_n,` *f, var1, var2, ..., varN*	read, skip non-numeric tokens
`rdline(`*f*`)`	returns next line from file *f*
`rdline(`*f, n*`)`	returns next *n* lines from file *f*
`sread,` *s, var1, var2, ..., varN*	reads the *varI* from string *s*

The data type and dimensions of the *varI* determine how the text is converted as it is read. The *varI* may be arrays, provided the arrays have identical dimensions. If the *varI* have length **L**, then the `read` is applied as if called **L** times, with successive elements of each of the *varI* read on each call.

The `read` function takes the `prompt` keyword to set the prompt string, which defaults to `"read> "`.

Both `read` and `sread` accept the `format` keyword. The format is a string containing conversion specifiers for the *varI*. The number of conversion specifiers should match the number of *varI*. If the *varI* are arrays, the format string is applied repeatedly until the arrays are filled.

Read format strings in Yorick have (nearly) the same meaning as the format strings for the ANSI standard C library `scanf` routine. In brief, a format string consists of:

- whitespace
  means to skip any number of whitespace characters in the source
- characters other than whitespace and `%`
  must match characters in the source exactly or the read operation stops
- conversion specifiers beginning with `%`
  each specifier ends with one of the characters `d` (decimal integer), `i` (decimal, octal, or hex integer), `o` (octal integer), `x` (hex integer), `s` (whitespace delimited string), any of `e`, `f`, or `g` (real), `[`*xxx*`]` to match the longest string of characters in the list, `[^`*xxx*`]` to match the longest string of characters not in the list, or `%` (the `%` character – not a conversion)

Writing Text

`write,` *f, expr1, expr2, .., exprN*	writes the *exprI* to file *f*
`write,` *expr1, expr2, .., exprN*	writes the *exprI* to terminal
`swrite(`*expr1, expr2, .., exprN*`)`	returns the *exprI* as a string

The `swrite` function returns an array of strings — one string for each line that would have been produced by the `write` function.

The *exprI* may be arrays, provided the arrays are conformable. In this case, the *exprI* are broadcast to the same length **L**, then the `write` is applied as if called **L** times, with successive elements of the *exprI* written on each call.

Both functions accept an optional `format` keyword. Write format strings in Yorick have (nearly) the same meaning as the format strings for the ANSI stacndard C library `printf` routine. In brief, a format string consists of:

- characters other than `%`
  which are copied directly to output
- conversion specifiers beginning with `%`
  of the general format `%`*FW.PSC* where:
  F is zero or more of the optional flags `-` (left justify), `+` (always print sign), (space) (leave space if +), `0` (leading zeroes)
  W is an optional decimal integer specifying the minimum number of characters to output
  .P is an optional decimal integer specifying the number of digits of precision
  S is one of `h`, `l`, or `L`, ignored by Yorick
  C is `d` or `i` (decimal integer), `o` (octal integer), `x` (hex integer), `f` (fixed point real), `e` (scientific real), `g` (fixed or scientific real), `s` (string), `c` (ASCII character), or `%` (the `%` character – not a conversion)

For example,

```
> write, format="   tp %7.4f %e\n", [1.,2.], [.5,.6]
   tp  1.0000 5.000000e-01
   tp  2.0000 6.000000e-01
>
```

Positioning a Text File

The `write` function always appends to the end of a file.

A sequence of `read` operations may be intermixed with `write` operations on the same file. The two types of operations do not interact.

The `read` and `rdline` functions read the file in complete lines; a file cannot be positioned in the middle of a line – although the `read` function may ignore a part of the last line read, subsequent `read` operations will begin with the next full line. The following functions allow the file to be reset to a previously read line.

`backup,` *f*	back up file *f* one line
m= `bookmark(`*f*`)`	record position of file *f* in *m*
`backup,` *f, m*	back up file *f* to *m*

The bookmark *m* records the current position of the file; it has a distinct Yorick data type, and the `info` or `print` function can be used to examine it. Without a bookmark, the `backup` function can back up only a single line.

Opening and Closing Binary Files

f= `openb(`*filename*`)`	open *filename* read-only
f= `updateb(`*filename*`)`	open *filename* read-write
f= `createb(`*filename*`)`	create the binary file *filename*
`close,` *f*	close file *f*

A binary file *f* has a Yorick data type which is distinct from a text file. The `info` and `print` functions describe *f*. The `close` function will be called implicitly when the last reference to a file disappears, e.g.– if *f* is redefined.

The data in a binary file is organized into named variables, each of which has a data type and dimensions. The `.` operator, which extracts members from a structure instance, accepts binary files for its left operand. Thus:

```
f= updateb("foo.bar");
print, f.var1, f.var2(2:8,::4);
f.var3(2,5)= 3.14;
close, f;
```

Opens a file, prints `var1` and a subarray of `var2`, sets one element of `var3`, then closes the file.

The `show` command prints an alphabetical list of the variables contained in a file:

`show,` *f*	shows the variables in file *f*
`show,` *f, pat*	show only names starting with *pat*
`get_vars(`*f*`)`	returns pointers to complete name lists for *f*

Saving and Restoring Variables

`save,` *f, var1, var2, ..., varN*	saves the *varI* in binary file *f*
`restore,` *f, var1, var2, ..., varN*	restores the *varI* from *f*
`save,` *f*	saves all array variables in binary file *f*
`restore,` *f*	restores all variables from binary file *f*

Unlike *f.varI= expr*, the `save` function will create the variable *varI* in the file *f* if it does not already exist.

The `restore` function redefines the in-memory *varI*. If several binary files are open simultaneously, the *f.varI* syntax will be more useful for reading variables than the `restore` function.

Note that a single command can be used to create a binary file, save variables *varI* in it, and close the file:

`save, createb(` *filename*`),` *var1, var2, ..., varN*

A similar construction using `restore` and `openb` is also useful.

Reading History Records

A binary file may have two groups of variables: those belonging to a set of history records, and non-record variables. The record variables may have different values in each record. The records are labeled by (optional) time and cycle numbers:

`jt,` *time*	advance all open record files to record nearest *time*
`jt,` *f, time*	advance file *f* to record nearest *time*
`jc,` *f, ncyc*	advance file *f* to record nearest *ncyc*
`get_times(`*f*`)`	return list of record times
`get_ncycs(`*f*`)`	return list of record cycles

Writing History Records

To write a family of files containing history records:

1. Create the file using `createb`.
2. Write all of the non-record (time independent) variables to the file using `save`.
3. Create a record which will correspond to time *time* and cycle *ncyc* for future `jt` and `jc` commands. Use:

`add_record,` *f, time, ncyc*	make new record at *time, ncyc*

4. Write all record (time dependent) variables to the file using `save`. After the first `add_record`, `save` will create and store record variables instead of non-record variables as in step 2.
5. Repeat steps 3 and 4 for each new record you wish to add to the file. For the second and subsequent records, `save` will not allow variables which were not written to the first record, or whose data type or shape has changed since the first record. That is, the structure of all history records in a file must be identical. Use type `pointer` variables to deal with data which changes in size, shape, or data type.

After each `add_record`, any number of `save` commands may be used to write the record.

If the current member of a history record file family has at least one record, and if the next record would cause the file to exceed the maximum allowed file size, `add_record` will automatically form the next member of the family. The maximum family member file size defaults to 4 MBytes, but:

`set_filesize,` *f, n_bytes*	set family member size

Opening Non-PDB Files

Yorick expects binary files to be in PDB format, but it can be trained to recognize any file whose format can be described using its Contents Log file description language. The basic idea is that if you can figure out how to compute the names, data types, dimensions, and disk addresses of the data in the file, you can train Yorick to open the file; once open, all of Yorick's machinery to manipulate the data will grind away as usual.

The following functions can be used to teach Yorick about a non-PDB file; use `help` to get complete details:

`_read,` *f, address, var*	raw binary read
`install_struct,` *f, struct_name, size, align, order, layout*	define a primitive data type
`add_variable,` *f, address, name, type, dimlist*	add a variable
`add_member,` *f, struct_name, offset, name, type, dimlist*	build up a data structure
`install_struct,` *f, struct_name*	finish `add_member` struct
`data_align,` *f, alignment*	specify default data alignment
`struct_align,` *f, alignment*	specify default struct alignment
`add_record,` *f, time, ncyc, address*	declare record
`add_next_file,` *f, filename*	open new family member

To write a plain text description of any binary file, use:

`dump_clog,` *f, clogname*	write Contents Log for *f*

Making Plots

`plg,` *y, x*	plot graph of 1-D *y* vs. *x*
`plm,` *mesh_args*	plot quadrilateral mesh
`plc,` *z, mesh_args*	plot contours of *z*
`plf,` *z, mesh_args*	plot filled mesh, filling with *z*
`plv,` *v, u, mesh_args*	plot vector field (*u,v*))
`pli,` *z, x0, y0, x1, y1*	plot image *z*
`pldj,` *x0, y0, x1, y1*	plot disjoint lines
`plt,` *text, x, y*	plot *text* at (*x,y*)

The *mesh_args* may be zero, two, or three arguments as follows:

- omitted to use the current default mesh set by:

`plmesh,` *mesh_args*	set default quadrilateral mesh
`plmesh`	delete current default quadrilateral mesh

- *y, x*
  To set mesh points to (*x, y*), which must be 2-D arrays of the same shape, with at least two elements in each dimension.
- *y, x, ireg*
  To set mesh points to (*x, y*), as above, with a region number array *ireg*. The *ireg* should be an integer array of the same shape as *y* and *x*, which has a non-zero "region number" for every meaningful zone in the problem. The first row and column of *ireg* do not correspond to any zone, since there are one fewer zones along each dimension than points in *y* and *x*.

The `plc` command accepts the `levs` keyword to specify the list of *z* values to be contoured; by default, eight linearly spaced levels are generated.

The `plc` and `plmesh` commands accept the `triangle` keyword to specify a detailed triangulation map for the contouring algorithm. Use the `help, triangle` for details.

The `plv` command accepts the `scale` keyword to specify the scaling factor to be applied to (*u, v*) before rendering the vectors in (*x, y*) space; by default, the vector lengths are chosen to be comparable to typical zone dimensions.

The `plm` command accepts the `boundary` keyword, which should be set to 1 if only the mesh boundary, rather than the mesh interior, is to be plotted.

The `plm`, `plc`, `plf`, and `plv` commands accept the `region` keyword to restrict the plot to only one region of the mesh, as numbered by the *ireg* argument. The default `region` is 0, which is interpreted to mean the every non-0 region of the mesh.

The `pli` command produces a cell array; the *x0, y0, x1, y1*, which are optional, specify the coordinates of the opposite corners of the cell array.

Numerous other keywords adjust the style of lines, text, etc.

Plot Paging and Hardcopy

`fma`	frame advance — next plot command will clear picture
`hcp`	send current picture to hardcopy file
`hcpon`	do automatic `hcp` at each `fma`
`hcpoff`	require explicit `hcp` for hardcopy
`hcp_out`	print and destroy current hardcopy file
`animate`	toggle animation mode (see `help`)

Setting Plot Limits

`logxy,` *xflag, yflag*	set log or linear axis scaling
`limits,` *xmin, xmax, ymin, ymax*	set plot limits
`limits,` *xmin, xmax*	set plot x-limits
`range,` *ymin, ymax*	set plot y-limits
l`= limits()`	save current plot limits in *l*
`limits,` *l*	restore plot limits saved in *l*

The four plot limits can be numbers to fix them at specific values, or the string `"e"` to specify extreme values. The `limits` command accepts the keywords `square`, `nice`, and `restrict`, which control how extreme values are computed.

Plot limits may also be set by point-and-click in the X window. The left button zooms in, middle button pans, and right button zooms out. Refer `help` on `limits` for details.

Managing Graphics Windows

`window,` *n*	switch to window *n* (0-7)
`winkill,` *n*	delete window *n* (0-7)

The `window` command takes several keywords, for example: `dpi=75` makes a smaller X window than the default `dpi=100`, `private=1` forces use of private instead of shared colors, `dump=1` forces the palette to be dumped to the `hcp` file, and `style` specifies an alternative style sheet for tick and label style (`"work.gs"` and `"boxed.gs"` are two predefined style sheets).

The `plf` and `pli` commands require a color palette:

`palette,` *name*	load the standard palette *name*
`palette,` *r, g, b*	load a custom palette
`palette, query=1,` *r, g, b*	retrieve current palette

Standard palette names: `"earth.gp"` (the default), `"gray.gp"`, `"yarg.gp"`, `"stern.gp"`, `"heat.gp"`, and `"rainbow.gp"`.

Graphics Query, Edit, and Defaults

`plq`	query (print) legends for current window
`plq,` *i*	query properties of element *i*
`pledit,` *key_list*	change properties of queried element
`pldefault,` *key_list*	set default window and element properties

The keywords which regulate the appearance of graphical primitives include (each has a `help` entry):

`legend`	string to use for legend
`hide`	non-zero to skip element
`type`	`"solid"`, `"dash"`, `"dot"`, `"dashdot"`, etc.
`width`	line width, default 1.0
`color`	`"fg"` (default), `"red"`, `"green"`, `"blue"`, etc.
`marks, marker, mspace, mphase, msize`	line markers
`rays, rspace, rphase, arroww, arrowl`	line ray arrows
`closed, smooth`	more line properties
`font, height, opaque, path, justify`	text properties
`hollow, aspect`	vector properties for `plv`