Yorick: An Interpreted Language

David H. Munro

This work was produced at the University of California, Lawrence Livermore National Laboratory under contract no. W-7405-ENG-48 between the U.S. Department of Energy and The Regents of the University of California for the operation of UC LLNL.

1 Basic Ideas

Yorick is an interpreted programming language. With Yorick you can read input (usually lists of numbers) from virtually any text or binary file, process it, and write the results back to another file. You can also plot results on your screen.

Yorick expression and flow control syntax is similar to the C programming language, but the Yorick language lacks declaration statements. Also, Yorick array index syntax more closely resembles Fortran than C.

1.1 Simple Statements

An interpreter immediately executes each line you type. Most Yorick input lines define a variable, invoke a procedure, or print the value of an expression.

1.1.1 Defining a variable

The following five Yorick statements define the five variables `c`, `m`, `E`, `theta`, and `file`:

```
c= 3.00e10;  m= 9.11e-28
E= m*c^2
theta= span(0, 6*pi, 200)
file= create("damped.txt")
```

Variable names are case sensitive, so `E` is not the same variable as `e`. A variable name must begin with a letter of the alphabet (either upper or lower case) or with an underscore (`_`); subsequent characters may additionally include digits.

A semicolon terminates a Yorick statement, so the first line contains two statements. To make Yorick statements easier to type, you don't need to put a semicolon at the end of most lines. However, if you are composing a Yorick program in a text file (as opposed to typing directly to Yorick itself, a semicolon at the end of every line reduces the chances of a misinterpretation, and makes your program easier to read.

Conversely, a new line need not represent the end of a statement. If the line is incomplete, the statement automatically continues on the following line. Hence, the second and third lines above could have been typed as:

```
E=
  m *
  c^2
theta= span(0, 6*pi,
            200)
```

In the second line, `*` and `^` represent multiplication and raising to a power. The other common arithmetic operators are `+`, `-`, `/` (division), and `%` (remainder or modulo). The rules for forming arithmetic expressions with these operators and parentheses are the same in Yorick as in Fortran or C (but note that `^` does not mean raise to a power in C, and Fortran uses the symbol `**` for that operation).

The `span` function returns 200 equally spaced values beginning with `0` and ending with `6*pi`. The variable `pi` is predefined as 3.14159...

The `create` function returns an object representing the new file. The variable `file` specifies where output functions should write their data. Besides numbers like `c`, `m`, and `E`, or arrays of numbers like `theta`, or files like `file`, Yorick variables may represent several other sorts of objects, taken up in later chapters.

The `=` operator is itself a binary operator, which has the side effect of redefining its left operand. It associates from right to left, that is, the rightmost `=` operation is performed first (all other binary operators except `^` associate from left to right). Hence, several variables may be set to a single value with a single statement:

```
psi= phi= theta= span(0, 6*pi, 200)
```

When you define a variable, Yorick forgets any previous value and data type:

```
phi= create("junk.txt")
```

1.1.2 Invoking a procedure

A Yorick function which has side effects may sensibly be invoked as a procedure, discarding the value returned by the function, if any:

```
plg, sin(theta)*exp(-theta/6), theta
write, file, theta, sin(theta)*exp(-theta/6)
close, file
```

The `plg` function plots a graph on your screen — in this case, three cycles of a damped sine wave. The graph is made by connecting 200 closely spaced points by straight lines.

The `write` function writes a two column, 200 line table of values of the same damped sine wave to the file '`damped.txt`'. Then `close` closes the file, making it unavailable for any further `write` operations.

A line which ends with a comma will be continued, to allow procedures with long argument lists. For example, the `write` statement could have been written:

```
write, file, theta,
             sin(theta)*exp(-theta/6)
```

A procedure may be invoked with zero arguments; several graphics functions are often used in this way:

```
hcp
fma
```

The `hcp` function writes the current graphics window contents to a "hardcopy file" for later retrieval or printing. The `fma` function stands for "frame advance" — subsequent plotting commands will draw on a fresh page. Normally, plotting commands such as `plg` draw on top of whatever has been drawn since the previous `fma`.

1.1.3 Printing an expression

An unadorned expression is also a legal Yorick statement; Yorick prints its value. In the preceding examples, only the characters you would type have been shown; to exhibit the `print` function and its output, I need to show you what your screen would look like — not only what you type, but what Yorick prints. To begin with, Yorick prompts you for input with a `>` followed by a space (see Section 1.3.5 [Prompts], page 23). In the examples in this section, therefore, the lines which begin with `>` are what you typed; the other line(s) are Yorick's responses.

```
> E
8.199e-07
> print, E
8.199e-07
> m;c;m*c^2
9.11e-28
3.e+10
```

```
8.199e-07
> span(0,2,5)
[0,0.5,1,1.5,2]
> max(sin(theta)*exp(-theta/6))
0.780288
```

In the span example, notice that an array is printed as a comma delimited list enclosed in square brackets. This is also a legal syntax for an array in a Yorick statement. For numeric data, the print function always displays its output in a format which would be legal in a Yorick statement; you must use the write function if you want "prettier" output. Beware of typing an expression or variable name which is a large array; it is easy to generate lots of output (you can interrupt Yorick by typing control-c if you do this accidentally, see Section 1.3.1 [Starting], page 16).

Most non-numeric objects print some useful descriptive information; for example, before it was closed, the file variable above would have printed:

```
> file
write-only text stream at:
  LINE: 201  FILE: /home/icf/munro/damped.txt
```

As you may have noticed, printing a variable and invoking a procedure with no arguments are syntactically indistinguishable. Yorick decides which operation is appropriate at run time. Thus, if file had been a function, the previous input line would have invoked the file function; as it was not a function, Yorick printed the file variable. Only an explicit use of the print function will print a function:

```
> print, fma
builtin fma()
```

1.2 Flow Control Statements

Almost everything you type at Yorick during an interactive session will be one of the simple Yorick statements described in the previous section – defining variables, invoking procedures, and printing expressions. However, in order to actually use Yorick you need to write your own functions and procedures. You can do this by typing Yorick statements at the keyboard, but you should usually put function definitions in a text file. Suggestions for how to organize such "include" files will be the topic of the next section. This section introduces the Yorick statements which define functions, conditionals, and loops.

1.2.1 Defining a function

Consider a damped sine wave. It describes the time evolution of an oscillator, such as a weight on a spring, which bobs up and down for a while after you whack it. The basic shape of the wave is determined by the Q of the oscillator; a high Q means there is little friction, and the weight will continue bobbing for many cycles, while a low Q means a lot of friction and few cycles. The amplitude of the oscillation is therefore a function of two parameters – the phase (time in units of the natural period of the oscillator), and the Q:

```
func damped_wave(phase, Q)
{
  nu= 0.5/Q;
  omega= sqrt(1.-nu*nu);
  return sin(omega*phase)*exp(-nu*phase);
}
```

Within a function body, I terminate every Yorick statement with a semicolon (see Section 1.1.1 [Define variable], page 1).

The variables `phase` and `Q` are called the parameters of the function. They and the variables `nu` and `omega` defined in the first two lines of the function body are *local* to the function. That is, calling `damped_wave` will not change the values of any variables named `phase`, `Q`, `nu`, or `omega` in the calling environment.

In fact, the only effect of calling `damped_wave` is to return its result, which is accomplished by the `return` statement in the third line of the body. That is, calling `damped_wave` has no side effects. You can use `damped_wave` in expressions like this:

```
> damped_wave(1.5, 3)
0.775523
> damped_wave([.5,1,1.5,2,2.5], 3)
[0.435436,0.705823,0.775523,0.659625,0.41276]
> q5= damped_wave(theta,5)
> fma; plg, damped_wave(theta,3), theta
> plg, damped_wave(theta,1), theta
```

The last two lines graphically compare Q=3 oscillations to Q=1 oscillations.

Notice that the arguments to `damped_wave` may be arrays. In this case the result will be the array of results for each element of input; hence `q5` will be an array of 200 numbers. Nor is Yorick confused by the fact that the `phase` argument (`theta`) is an array, while the `Q` argument (`5`) is a scalar. The precise rules for "conformability" between two arrays will be described later (see Section 2.6 [Broadcasting], page 45); usually you get what you expected.

In this case, as Yorick evaluates `damped_wave`, `nu` and `omega` will both be scalars, since `Q` is a scalar. On the other hand, `omega*phase` and `nu*phase` become arrays, since `phase` is an array. Whenever an operand is an array, an arithmetic operation produces an array as its result.

1.2.2 Defining Procedures

Any Yorick function may be invoked as a procedure; the return value is simply discarded. Calling `damped_wave` as a procedure would be pointless, since it has no side effects. The converse case is a function to be called *only* for its side effects. Such a function need not have an explicit `return` statement:

```
func q_out(Q, file_name)
{
  file= create(file_name);
  write, file, "Q= "+pr1(Q);
  write, file, "   theta        amplitude";
  write, file, theta, damped_wave(theta,Q);
}
```

The `pr1` function (for "print one value") returns a string representation of its numeric argument, and the `+` operator between two strings concatenates them.

You would invoke `q_out` with the line:

```
q_out, 3, "q.out"
```

Besides lacking an explicit `return` statement, the `q_out` function has two other peculiarities worth mentioning:

First, the variable `theta` is never defined. But neither are the functions `create` and `write`. Any symbol not defined within a function is *external* to the function. As already noted, parameters and variables defined within the function are *local* to the function. Here, `theta` is the 200 element array of phase angles defined before q_out was called. You can change theta (both its dimensions and its values) before the next call to `q_out`; that is, an external reference may change between calls to a function that uses it.

Second, `file` is never explicitly closed. When a function returns, its local variables (such as `file`) disappear; when a file object disappears, the associated file automatically closes.

1.2.3 Conditional Execution

The design of the `q_out` function can be improved. As written, each output file will contain only a single wave. You might want the option of writing several waves into a single file. Consider this alternative:

```
func q_out(Q, file)
{
  if (!is_stream(file)) file= create(file);
  write, file, "Q= "+pr1(Q);
  write, file, "   theta       amplitude";
  write, file, theta, damped_wave(theta,Q);
  return file;
}
```

The `if` statement executes its body (the redefinition of `file`) if and only if its condition (`!is_stream(file)`) is true. Any scalar number may serve as a condition – a non-zero value is "true", and the value zero is "false".

The `is_stream` function returns 1 (true) when its argument is a file object (a "data stream"), and 0 (false) otherwise. In particular, if `file` is a text string (like `"q.out"`), `is_stream` returns 0. The unary operator `!` is logical negation, that is, "not".

Hence, if the `file` argument is not already a file object, the new `q_out` presumes it is the name of a file, which it creates, redefining `file` as the associated file object. Thus, after the first line of the function body, `file` will be a file object, even if a file name was passed into the function. Furthermore, since the parameter `file` is local to `q_out`, none of this hocus pocus will have any effect outside `q_out`.

The second trick in the new `q_out` is the reappearance of a `return` statement. The original calling sequence:

```
q_out, 3, "q.out"
```

has the same result as before – the `if` condition is true, so the file is created, then the wave data is written. This time the file object is returned, only to be discarded because `q_out` was invoked as a procedure. When the file object disappears, the file closes. But if `q_out` were invoked as a function, the file object can be saved, which keeps the file open:

```
f= q_out(3,"q.out")
q_out,2,f
q_out,1,f
close,f
```

Now the file '`q.out`' contains the Q=3 wave, followed by the Q=2 and Q=1 waves. In the second and third calls to `q_out`, the `file` parameter is already a file object, so the `if` condition is false, and `create` is not called. Notice that the file does not close when the return value from the second (or third) call is discarded; the variable `f` refers to the same file object as the discarded return value. Without an explicit call to `close`, a file only closes when the *final* reference to it disappears.

1.2.3.1 General `if` and `else` constructs

The most general form of the `if` statement is:

```
if (condition) statement_if_true;
else statement_if_false;
```

When you need to choose among several alternatives, make the `else` clause itself an `if` statement:

```
if (condition_1) statement_if_1;
else if (condition_2) statement_if_2;
else if (condition_3) statement_if_3;
...
else statement_if_none;
```

The final `else` is always optional – if it is not present, nothing at all happens if none of the conditions is satisfied.

Often you need to execute more than one statement conditionally. Quite generally in Yorick, you may group several statements into a single compound statement by means of curly braces:

```
{ statement_1; statement_2; statement_3; statement_4; }
```

Ordinarily, both curly braces and each of the statements should be written on a separate line, with the statements indented to make their membership in the compound more obvious. Further, in an if-else construction, when one branch requires more than one statement, you should write every branch as a compound statement with curly braces. Therefore, for a four branch `if` in which the second `if` requires two statements, and the `else` three, write this:

```
if (condition_1) {
  statement_if_1;
} else if (condition_2) {
  statement_if_2_A;
  statement_if_2_B;
} else if (condition_3) {
  statement_if_3;
} else {
  statement_if_none_A;
  statement_if_none_B;
  statement_if_none_C;
}
```

The `else` statement has a very unusual syntax: It is the only statement in Yorick which depends on the form of the *previous* statement; an `else` must follow an `if`. Because Yorick statements outside functions (and compound statements) are executed immediately, you must be very careful using `else` in such a situation. If the `if` has already been executed (as it would be in the examples without curly braces), the following `else` leads to a syntax error!

The best way to avoid this puzzling problem is to always use the curly brace syntax of the latest example when you use `else` outside a function. (You don't often need it in such a context anyway.)

1.2.3.2 Combining conditions with `&&` and `||`

The logical operators `&&` ("and") and `||` ("or") combine conditional expressions; the `!` ("not") operator negates them. The `!` has the highest precedence, then `&&`, then `||`; you will need parentheses to force a different order.

(Beware of the bitwise "and" and "or" operators `&` and `|` – these should *never* be used to combine conditions; they are for set-and-mask bit fiddling.)

The operators for comparing numeric values are `==` (equal), `!=` (not equal), `>` (greater), `<` (less), `>=` (greater or equal), and `<=` (less or equal). These all have higher precedence than `&&`, but lower than `!` or any arithmetic operators.

```
if ((a<b && x>a && x<b) || (a>b && x<a && x>b))
  write, "x is between a and b"
```

Here, the expression for the right operand to `&&` will execute only if its left operand is actually true. Similarly, the right operand to `||` executes only if its left proves false. Therefore, it is important to order the operands of `&&` and `||` to put the most computationally expensive expression on the right — even though the logical "and" and "or" functions are commutative, the order of the operands to `&&` and `||` can be critical.

In the example, if `a>b`, the `x>a` and `x<b` subexpressions will not actually execute since `a<b` proved false. Since the left operand to `||` was false, its right operand will be evaluated.

Despite the cleverness of the `&&` and `||` operators in not executing the expression for their right operands unless absolutely necessary, the example has obvious inefficiencies: First, if `a>=b`, then both `a<b` and `a>b` are checked. Second, if `a<b`, but `x` is not between `a` and `b`, the right operand to `||` is evaluated anyway. Yorick has a ternary operator to avoid this type of inefficiency:

```
expr_A_or_B= (condition? expr_A_if_true : expr_B_if_false);
```

The `?:` operator evaluates the middle expression if the condition is true, the right expression otherwise. Is that so? Yes : No. The efficient betweeness test reads:

```
if (a<b? (x>a && x<b) : (x<a && x>b))
  write, "x is between a and b";
```

1.2.4 Loops

Most loops in Yorick programs are implicit; remember that operations between array arguments produce array results. Whenever you write a Yorick program, you should be suspicious of *all* explicit loops. Always ask yourself whether a clever use of array syntax could have avoided the loop.

To illustrate an appropriate Yorick loop, let's revise `q_out` to write several values of Q in a single call. Incidentally, this sort of incremental revision of a function is very common in Yorick program development. As you use a function, you notice that the surrounding code is often the same, suggesting a savings if it were incorporated into the function. Again, a careful job leaves all of the previous behavior of `q_out` intact:

```
func q_out(Q, file)
```

```
{
  if (!is_stream(file)) file= create(file);
  n= numberof(Q);
  for (i=1 ; i<=n ; ++i) {
    write, file, "Q= "+pr1(Q(i));
    write, file, "   theta       amplitude";
    write, file, theta, damped_wave(theta,Q(i));
  }
  return file;
}
```

Two new features here are the `numberof` function, which returns the length of the array `Q`, and the array indexing syntax `Q(i)`, which extracts the `i`-th element of the array `Q`. If `Q` is scalar (as it had to be before this latest revision), `numberof` returns 1, and `Q(1)` is the same as `Q`, so scalar `Q` works as before.

But the most important new feature is the `for` loop. It says to initialize `i` to 1, then, as long as `i` remains less than or equal to `n`, to execute the loop body, which is a compound of three write statements. Finally, after each pass, the increment expression `++i` is executed before the `i<=n` condition is evaluated. `++i` is equivalent to `i=i+1`; `--i` means `i=i-1`.

A loop body may also be a single statement, in which case the curly braces are unecessary. For example, the following lines will write the Q=3, Q=2, and Q=1 curves to the file ‘`q.out`’, then plot them:

```
q_list= [3,2,1]
q_out, q_list, "q.out"
fma; for(i=1;i<=3;++i) plg, damped_wave(theta,q_list(i)), theta
```

A `for` loop with a `plg` body is the easiest way to overlay a series of curves. After the three simple statement types, `for` statements see the most frequent direct use from the keyboard.

1.2.4.1 The `while` and `do while` statements

The `while` loop is simpler than the `for` loop:

```
while (condition) body_statement
```

The `body_statement` — very oten a compound statement enclosed in curly braces — executes over and over until the `condition` becomes false. If the `condition` is false to begin with, `body_statement` never executes.

Occasionally, you want the `body_statement` to execute and set the `condition` *before* it is tested for the first time. In this case, use the `do while` statement:

```
do {
  body_statement_A;
  body_statement_B;
  ...etc...
} while (condition);
```

1.2.4.2 The `for` statement

The `for` statement is a "packaged" form of a `while` loop. The meaning of this generic `for` statement is the same as the following `while`:

```
for (start_statements ; condition ; step_statements) body_statements

start_statments
while (condition) {
  body_statements
  step_statements
}
```

There only two reasons to prefer `for` over `while`: Most importantly, `for` can show "up front" how a loop index is initialized and incremented, rather than relegating the increment operation (`step_statements`) to the end of the loop. Secondly, the `continue` statement (see Section 1.2.4.3 [goto], page 13) branches just past the body of the loop, which would include the `step_statements` in a `while` loop, but not in a `for` loop.

In order to make Yorick loop syntax agree with C, Yorick's `for` statement has a syntactic irregularity: If the `start_statements` and `step_statements` consist of more than one statement, then commas (not semicolons) separate the statements. (The C comma operator is not available in any other context in Yorick.) The two semicolons separate the start, condition, and step clauses and must always be present, but any or all of the three clauses themselves may be blank. For example, in order to increment two variables `i` and `j`, a loop might look like this:

```
for (i=1000, j=1 ; i>j ; i+=1000, j*=2);
```

This example also illustrates that the `body_statements` may be omitted; the point of this loop is merely to compute the first `i` and `j` for which the condition is not satisfied. The trailing semicolon is necessary in this case, since otherwise the line would be continued (on the assumption that the loop body was to follow).

The `+=` and `*=` are special forms of the `=` operator; `i=i+1000` and `j=j*2` mean the same thing. Any binary operator may be used in this short-hand form in order to increment a variable. Like `++i` and `--i`, these are particularly useful in loop increment expressions.

1.2.4.3 Using break, continue, and goto

The `break` statement jumps out of the loop containing it. The `continue` statement jumps to the end of the loop body, "continuing" with the next pass through the loop (if any). In deeply nested loops — which are extremely rare in Yorick programs — you can jump to more general places using the `goto` statement. For example, here is how to break out or continue from one or both levels of a doubly nested loop:

```
 for (i=1 ; i<=n ; ++i) {
   for (j=1 ; j<=m ; ++j) {
     if (skip_to_next_j_now) continue;
     if (skip_to_next_i_now) goto skip_i;
     if (break_out_of_inner_loop_now) break;
     if (break_out_of_both_loops_now) goto done;
     more_statements_A;
   }
   more_statements_B;
  skip_i:
 }
done:
 more_statements_C;
```

The `continue` jumps just after `more_statements_A`, the `break` just before `more_statements_B`.

Break and continue statements may be used to escape from `while` or `do while` loops as well.

If `while` tests the condition before the loop body, and `do while` checks after the loop body, you may have wondered what to do when you need to check the condition in the middle of the loop body. The answer is the "do forever" construction, plus the `break` statement (note the inversion of the sense of the condition):

```
for (;;) {
  body_before_test;
  if (!condition) break;
  body_after_test;
}
```

1.2.5 Variable scope

By default, dummy parameters and variables which are defined in a function body before any other use are *local* variables. Variables (or functions) used before their definition are *external* variables (see Section 1.2.2 [Define procedure], page 6, see Section 1.2.1 [Define function], page 5). A variable (or function) defined outside of all functions is just that — external to all functions and local to none.

Whenever a function is called, Yorick remembers the external values of all its local variables, then replaces them by their local values. Thus, all of its local variables are potentially "visible" as external variables to any function it calls. When it returns, the function replaces all its local variables by the values it remembered. Neither that function, nor any function it calls can affect these remembered values; Yorick provides no means of "unmasking" a local variable.

The default rule for determining whether a variable should have local or external scope fails in two cases: First, you may want to redefine an external variable without looking at its value. Second, a few procedures set the values of their parameters when they return; it may appear to Yorick's parser that such a variable has been used without being defined, even though you intend it to be local to the function. The `extern` and `local` statements solve these two problems, respectively.

1.2.5.1 `extern` statements

Suppose you want to write a function which sets the value of the `theta` array used by all the variants of the `q_out` function:

```
func q_out_domain(start, stop, n)
{
  extern theta;
  theta= span(start, stop, n);
}
```

Without the `extern` statement, `theta` will be a local variable, whose external value is restored when `q_out_domain` returns (thus, the function would be a no-op). With the `extern` statement, q_out_domain has the intended side effect of setting the value of theta. The new theta would be used by subsequent calls to q_out.

Any number of variables may appear in a single extern statement:

```
extern var1, var2, var3, var4;
```

1.2.5.2 local statements

The save and restore functions store and retrieve Yorick variables in self-descriptive binary files (called PDB files). To create a binary file 'demo.pdb', save the variables theta, E, m, and c in it, then close the file:

```
f= createb("demo.pdb")
save, f, theta, E, m, c
close, f
```

To open this file and read back the theta variable:

```
restore, openb("demo.pdb"), theta, c
```

For symmetry with save, the restore function has the unusual property that it redefines its parameters (except the first). Thus, theta is redefined by this restore statement, just as it would have been by a theta= statement. The Yorick parser does not understand this, which means that you must be careful when you place such a function call inside a function:

```
func q_out_file(infile, Q, outfile)
{
  if (!is_stream(infile)) infile= openb(infile);
  local theta;
  restore, infile, theta;
  return q_out(Q, outfile);
}
```

This variant of q_out uses the theta variable read from infile, instead of an external value of theta. Without the local declaration, restore would clobber the external value of theta. This way, when q_out_file calls q_out, it has remembered the external value of theta, but q_out "sees" the value of theta restored from infile. When q_out_file finally returns, it replaces the original value of theta intact.

Any number of variables may appear in a single `local` statement:

```
local var1, var2, var3, var4;
```

1.3 The Interpreted Environment

The Yorick program accepts only complete input lines typed at your keyboard. Typing a command to Yorick presumes a "command line interface" or a "terminal emulator" which is not a part of Yorick. I designed Yorick on the assumption that you have a good terminal emulator program. In particular, Yorick is much easier to use if you can recall and edit previous input lines; as in music, repitition with variations is at the heart of programming. My personal recommendation is shell-mode in GNU Emacs.

Therefore, Yorick inherits most of its "look and feel" from your terminal emulator. Yorick's distinctive prompts and error messages are described later in this section.

Any significant Yorick program will be stored in a text file, called an *include* file, after the command which reads it. Use your favorite text editor to create and modify your include files. Again, GNU Emacs is my favorite — use its c-mode to edit Yorick include files as well as C programs. Just as C source file names should end in '`.c`' or '`.h`', and Fortran source file names should end in '`.f`', so Yorick include file names should end in '`.i`'.

This section begins with additional stylistic suggestions concerning include files. In particular, Yorick's `help` command can find documentation comments in your include files if you format them properly. All of the built-in Yorick functions, such as `sin`, `write`, or `plg`, come equipped with such comments.

1.3.1 Starting, stopping, and interrupting Yorick

Start Yorick by typing its name to your terminal emulator. When you are done, use the `quit` function to exit gracefully:

```
% yorick
 Yorick ready.  For help type 'help'.
> quit
%
```

You can interrupt Yorick at any time by typing Control-C; this causes an immediate runtime error (see Section 1.3.7 [Errors], page 25). (Your terminal emulator and operating system must be able to send Yorick a SIGINT signal in order for this to work. On UNIX systems, you can set this character using the `intr` option of the `stty` command; Control-C is the usual setting.)

1.3.2 Include files

When you run Yorick in a window or Emacs environment, keep a window containing the include file you are writing, in addition to the window where Yorick is running. When you want to test the latest changes to your include file — let's call it '`damped.i`' — save the file, then move to your Yorick window and type:

```
#include "damped.i"
```

The special character `#` reminds you that the include directive is not a Yorick statement, like all other inputs to Yorick. Instead, `#include` directs Yorick to switch its input stream from the keyboard to the quoted file. Yorick then treats each line of the file exactly as if it were a line you had typed at the keyboard (including, of course, additional `#include` lines). When there are no more lines in the file, the input stream switches back to the keyboard.

While you are debugging an include file, you will ordinarily include it over and over again, until you get it right. This does no harm, since any functions or variables defined in the file will be replaced by their new definitions the next time you include the file, just as they would if you typed new definitions.

Ideally, your include files should consist of a series of function definitions and variable initializations. If you follow this discipline, you can regard each include file as a library of functions. You type `#include` once in order to load this library, making the functions defined there available, just like Yorick's built in functions. Many such libraries come with Yorick — Bessel functions, cubic spline interpolators, and other goodies.

1.3.2.1 A sample include file

Here is '`damped.i`', which defines the damped sine wave functions we've been designing in this chapter:

```
/* damped.i */
```

```
local damped;
/* DOCUMENT damped.i --- compute and output damped sine waves
   SEE ALSO: damped_wave, q_out
 */

func damped_wave(phase, Q)
/* DOCUMENT damped_wave(phase, Q)
     returns a damped sine wave evaluated at PHASE, for quality factor Q.
     (High Q means little damping.)  The PHASE is 2*pi after one period of
     the natural frequency of the oscillator.  PHASE and Q must be
     conformable arrays.

   SEE ALSO: q_out
 */
{
  nu= 0.5/Q;
  omega= sqrt(1.-nu*nu);
  return sin(omega*phase)*exp(-nu*phase);  /* always zero at phase==0 */
}

func q_out(Q, file)
/* DOCUMENT q_out, Q, file
         or q_out(Q, file)
     Write the damped sine wave of quality factor Q to the FILE.
     FILE may be either a filename, to create the file, or a file
     object returned by an earlier create or q_out operation.  If
     q_out is invoked as a function, it returns the file object.

     The external variable
          theta
     determines the phase points at which the damped sine wave is
     evaluated; q_out will write two header lines, followed by
     two columns, with one line for each element of the theta
     array.  The first column is theta; the second is
     damped_wave(theta, Q).

     If Q is an array, the two header lines and two columns will
     be repeated for each element of Q.

   SEE ALSO: damped_wave
 */
```

```
{
  if (!is_stream(file)) file= create(file); /* file name --> object */
  n= numberof(Q);
  for (i=1 ; i<=n ; ++i) { /* loop on elements of Q */
    write, file, "Q= "+pr1(Q(i));
    write, file, "   theta      amplitude";
    write, file, theta, damped_wave(theta,Q(i));
  }
  return file;
}
```

You would type

```
#include "damped.i"
```

to read the Yorick statements in the file. The first thing you notice is that there are comments — the text enclosed by `/* */`. If you take the trouble to save a program in an include file, you will be wise to make notes about what it's for and annotations of obscure statements which might confuse you later. Writing good comments is arguably the most difficult skill in programming. Practice it diligently.

1.3.2.2 Comments

There are two ways to insert comments into a Yorick program. These are the C style `/* ... */` and the C++ style `//`:

```
// C++ style comments begin with // (anywhere on a line)
// and end at the end of that line.

E= m*c^2;   /* C style comments begin with slash-star, and
               do not end until start-slash, even if that
               is several lines later.  */
/* C style comments need not annotate a single line.
 * You should pick a comment style which makes your
 * code attractive and easy to read.  */

F= m*a;           // Here is another C++ style comment...
divE= 4*pi*rho;   /* ... and a final C style comment. */
```

I strongly recommend C++ style comments when you "comment out" a sequence of Yorick statements. C style comments do not nest properly, so you can't comment out a series of lines which contain comments:

```
/*
E= m*c^2;   /* ERROR -- this ends the outer comment --> */
F= m*a
*/ <-- then this causes a syntax error
```

The C++ style not only works correctly; it also makes it more obvious that the lines in question are comments:

```
// E= m*c^2;   /* Any kind of comment could go here.  */
// F= m*a;
```

Yorick recognizes one special comment: If the first line of an include file begins with `#!`, Yorick ignores that line. This allows Yorick include scripts to be executable on UNIX systems supporting the “pound bang” convention:

```
#!/usr/local/bin/yorick -batch
/* If this file has execute permission, UNIX will use Yorick to
 * execute it.  The Yorick function get_argv can be used to accept
 * command line arguments (see help, get_argv).  You might want
 * to use -i instead of -batch on the first line.  Read the help
 * on process_argv and batch for more information.  */
write, "The square root of pi is", sqrt(pi);
quit;
```

1.3.2.3 `DOCUMENT` comments

Immediately after (within a few lines of) a `func`, `extern`, or `local` statement (see Section 1.2.5 [Scoping], page 14), you may put a comment which begins with the eleven characters `/* DOCUMENT`. Although Yorick itself doesn't pay any more attention to a `DOCUMENT` comment than to any other comment, there is a function called `help` which does. What Yorick does do when it sees a `func`, `extern`, or `local` (outside of any function body), is to record the include file name and line number where that function or variable(s) are defined.

Later, when you ask for help on some topic, the `help` function asks Yorick whether it knows the file and line number where that topic (a function or variable) was defined. If so, it opens the file, goes to the line number, and scans forward a few lines looking for a `DOCUMENT` comment. If it finds one, it prints the entire comment.

The file `damped.i` has three `DOCUMENT` comments: one for a fictitious variable called `damped`, so that someone who saw the file but didn't know the names of the functions inside could find out, and one each for the `damped_wave` and `q_out` functions.

Near the end of most DOCUMENT comments, you will find a SEE ALSO: field which feeds the reader the names of related functions or variables which also have DOCUMENT comments. If you use these wisely, you can lead someone (often an older self) to all of the documentation she needs to be able to use your package.

This low-tech form of online documentation is surprisingly effective: easy to create, maintain, and use. As an automated step toward a more formal document, the mkdoc function (in the Yorick library include file 'mkdoc.i') collects all of the DOCUMENT comments in one or more include files, alphabetizes them, adds a table of contents, and writes them into a file suitable for printing.

1.3.2.4 Where Yorick looks for include files

You can specify a complete path name (including directories) for the file in an #include directive. More usually, you will use a relative path name. In that case, Yorick tries to find the file relative to these four directories, in this order:

1. Your current working directory.
2. '~/Yorick', that is, the 'Yorick' subdirectory of your home directory.
3. 'Y_SITE/include', where 'Y_SITE' is the directory where Yorick was installed at your site (help will tell you where this is).
4. 'Y_SITE/contrib'

You can use the set_path command in your 'custom.i' file in order to change this path, but you should be very cautious if you do this.

The '~/Yorick' directory is where you put all of the Yorick include files you frequently use, which have not been placed in the include or contrib directories at your site. You can also override an include file in one of these places by placing a file of the same name in your '~/Yorick' directory.

1.3.2.5 The 'custom.i' file

When Yorick starts, the last thing it does before prompting you is to include the file 'custom.i'. The default 'custom.i' is in the 'Y_SITE/include/custom.i'. (Yorick's help command will tell you the actual name of the 'Y_SITE' directory at your site.)

If you create the directory '~/Yorick', you can place your own version of `custom.i` there to override the default. Always begin by copying the default file to your directory. Then add your customizations to the bottom of the default file. Generally, these customizations consist of `#include` directives for function libraries you use heavily, plus commands like `pldefault` to set up your plotting style preferences.

1.3.3 The `help` function

Use the `help` function to get online help. Yorick will parrot the associated `DOCUMENT` comment to your terminal:

```
> #include "damped.i"
> help, damped
  DOCUMENT damped.i --- compute and output damped sine waves
  SEE ALSO: damped_wave, q_out
defined at:   LINE: 3  FILE: /home/icf/munro/damped.i
>
```

Every Yorick function (including `help`!) has a `DOCUMENT` comment which describes what it does. Most have `SEE ALSO:` references to related help topics, so you can navigate your way through a series of related topics.

Whenever you want more details about a Yorick function, the first thing to do is to see what `help` says about it. Note that, in addition to the `DOCUMENT` comment, `help` also reports the full file name and line number where the function is defined. That way, if the comment doesn't tell you what you need to know, you can go to the file and read the complete definition of the function.

The help for `help` itself, which is the default topic, is of particular interest, even to a Yorick expert: it tells you the directory where you can find the include files for all of Yorick's library functions. Just type `help`, like it says when Yorick starts. One of the things you will find in Yorick's directory tree is a '`doc`' directory, which contains not only the source for this manual, but also alphabetized listings of all `DOCUMENT` comments. Read the `README` file in the '`doc`' directory.

1.3.4 The `info` function

If you want to know the data type and dimensions of a variable, use the `info` function. Unlike `help`, which is intended to tell you what a thing *means*, `info` simply tells what a thing *is*.

```
> info, theta
 array(double,200)
> info, E
 array(double)
```

Here, `double` means “double precision floating point number”, which is the default data type for any real number. The default integer data type is called `long`. (Both these names come from the C language, which gives them a precise meaning.)

Notice that the scalar value `E` is, somewhat confusingly, called an “array”. In fact, a scalar is a special case of an array with zero dimensions. The `info` function is designed to print a Yorick expression which will create a variable of the same data type and shape as its argument. Thus, `array(double,200)` in an expression would evaluate to an array of 200 real numbers, while `array(double)` is a real scalar (the values are always zero).

Using `info` on a non-numeric quantity (a file object, a function, etc.) results in the same output as the `print` function. If an array of numbers might be large, try `info` before `print`.

1.3.5 Prompts

Yorick occasionally prompts you with something other than `>`. The usual `>` prompt tells you that Yorick is waiting for a new input line. The other prompts alert you to unusual situations:

`cont>`
: The previous input line was not complete; Yorick is waiting for its continuation before it does anything. Type several close brackets or control-C if you don’t want to complete the line.

`dbug>`
: When an error occurs during the execution of a Yorick program, Yorick offers you the choice of entering “debug mode”. In this mode, you are “inside” the function where the error occurred. You can type any Yorick statement you wish, but until you explicitly exit from debug mode by means of the `dbexit` function, Yorick will prompt you with `dbug>` instead of its usual prompt. (see Section 1.3.7.2 [Simple debugging], page 26)

`quot>`
: It is possible to continue a long string constant across multiple lines. If you do, this will be your prompt until you type the closing `"`. Don’t ever do this intentionally; use the string concatenation operator `+` instead, and break long strings into pieces that will fit on a single line.

`comm>`
: For some reason, you began a comment on the previous line, and Yorick is waiting for the `*/` to finish the comment. I can’t imagine why you would put comments in what you were typing at the terminal.

1.3.6 Shell commands, removing and renaming files

Yorick has a `system` function which you can use to invoke operating system utilities. For example, typing `ls *.txt` to a UNIX shell will list all files in your current working directory ending with '`.txt`'. Similarly, `pwd` prints the name of your working directory:

```
> system, "pwd"
/home/icf/munro
> system, "ls *.txt"
   damped.txt      junk.txt
```

This is so useful that there is a special escape syntax which automatically generates the system call, so you don't need to type the name `system` or all of the punctuation. The rule is, that if the first character of a Yorick statement is `$` (dollar), the remainder of that line becomes a quoted string which is passed to the `system` function. Hence, you really would have typed:

```
> $pwd
/home/icf/munro
> $ls *.txt
   damped.txt      junk.txt
```

Note that you cannot use `;` to stack `$` escaped lines — the semicolon will be passed to `system`. Obviously, this syntax breaks all of the ordinary rules of Yorick's grammar.

On UNIX systems, the `system` function (which calls the ANSI C standard library function of the same name) usually executes your command in a Bourne shell. If you are used to a different shell, you might be surprised at some of the results. If you have some complicated shell commands for which you need, say, the C-shell `csh`, just start a copy of that shell before you issue your commands:

```
> $csh
% ls *.txt
  damped.txt    junk.txt
% exit
>
```

As this example shows, you can start up an interactive program with the `system` function. When you exit that program, you return to Yorick.

One system command which you cannot use is `cd` (or `pushd` or `popd`) to change directories. The working directory of the shell you start under Yorick will change, but the change has no effect on Yorick's own working directory. Instead, use Yorick's own `cd` function:

```
> cd, "new/working/directory"
>
```

Also, if you write a Yorick program which manipulates temporary files, you should not use the UNIX commands `mv` or `rm` to rename or remove the files; any time you use the `system` command you are restricting your program to a particular class of operating systems. Instead, use Yorick's `rename` and `remove` functions, which will work under any operating system.

1.3.7 Error Messages

When Yorick cannot decipher the meaning of a statement, you have made a *syntax* error. Syntax errors are mostly simple typos, but a mechanical language like Yorick can be exasperatingly picky:

```
> th= span(0,2*pi,200)
> for (i=1 ; i<=5 ; ++i) { r=cos(i*th); plg,r*sin(th),r*cos(th) }
SYNTAX: parse error near }
>
```

The only mistake here is that Yorick wants a semicolon (or newline) before the close curly brace completing a compound statement. If you think “fixing” this type of behavior would be simple, I suggest you study parsers. Every programming language has its quirks.

When Yorick detects a syntax error, the error message always begins with `SYNTAX:`. The entire block containing the error will be discarded; no statements will be executed which might lead to the execution of the statement containing the error. If the error is in a function body, the function will not be defined. However, if the sytax error occurs reading an include file, Yorick continues to parse the file looking for additional syntax errors. After about a dozen syntax errors in a single file, Yorick gives up and waits for keyboard input. Therefore, you may be able to repair several syntax errors before you re-include the file.

All other errors are *runtime* errors.

1.3.7.1 Runtime errors

Runtime errors in Yorick are often simple typos, like syntax errors:

```
> theta= span(0,2*pi,200)
> wave= sin(thta)*exp(-0.5*theta)
```

```
ERROR (*main*) expecting numeric argument
WARNING source code unavailable (try dbdis function)
now at pc= 1 (of 23), failed at pc= 5
 To enter debug mode, type <RETURN> now (then dbexit to get out)
>
```

Many errors of this sort would be detected as syntax errors if you had to declare Yorick variables. Yorick's free-and-easy attitude toward declaration of variables is particularly annoying when the offending statement is in a conditional branch which is very rarely executed. When a bug like that ambushes you, be philosophical: Minutely declared languages will just ambush you in more subtle ways.

Other runtime errors are more interesting; often such a bug will teach you about the algorithm or even about the physical problem:

```
> #include "damped.i"
> theta= span(0, 6*pi, 300)
> amplitude= damped_wave(theta, 0.25)
ERROR (damped_wave) math library exception handler called
  LINE: 19  FILE: /home/icf/munro/damped.i
 To enter debug mode, type <RETURN> now (then dbexit to get out)
>
```

What is an oscillator with a Q of less than one half? Maybe you don't care about the so-called *overdamped* case — you really wanted Q to be 2.5, not 0.25. On the other hand, maybe you need to modify the `damped_wave` function to handle the overdamped case.

1.3.7.2 How to respond to a runtime error

When Yorick stops with a runtime error, you have a choice: You can either type the next statment you want to execute, or you can type a carriage return (that is, a blank line) to enter debug mode. The two possibilities would look like this:

```
ERROR (damped_wave) math library exception handler called
  LINE: 19  FILE: /home/icf/munro/damped.i
 To enter debug mode, type <RETURN> now (then dbexit to get out)
> amplitude= damped_wave(theta, 2.5)
>

ERROR (damped_wave) math library exception handler called
  LINE: 19  FILE: /home/icf/munro/damped.i
 To enter debug mode, type <RETURN> now (then dbexit to get out)
```

```
>
dbug>
```

In the second case, you have entered debug mode, and the `dbug>` prompt appears. In debug mode, Yorick leaves the function which was executing and its entire calling chain intact. You can type any Yorick statement; usually you will print some values or plot some arrays to try to determine what went wrong. When you reference or modify a variable which is local to the function, you will "see" its local value:

```
dbug> nu; 1-nu*nu
2
-3
dbug>
```

As soon as possible, you should escape from debug mode using the `dbexit` function:

```
dbug> dbexit
>
```

You may also be able to repair the function's local variables and resume execution. To modify the value of a variable, simply redefine it with an ordinary Yorick statement. The `dbcont` function continues execution, beginning by re-executing the statement which failed to complete. Use the `help` function (see Section 1.3.3 [Help], page 22) to learn about the other debugging functions; the help for the `dbexit` function describes them all.

2 Using Array Syntax

Most Yorick statements look like algebraic formulas. A variable name is a string like `Var_1` — upper or lower case characters (case matters), digits, or underscores in any combination except that the first character may not be a digit. Expressions consist of the usual arithmetic operations `+ - * /`, with parentheses to indicate the order of operations (when that order is different than or unclear from the ordinary rules of precedence in algebra). Elementary mathematical functions such as `exp(x)`, `cos(x)`, or `atan(x)` look just like that.

Usually, a Yorick variable is a parametric representation of a mathematical function. The variable is an array of numbers which are values of the function at a number of points; few points to represent the function coarsely, more for an accurate rendition. The parameters of the function are the indices into the array, which rarely make an explicit appearance in Yorick programs. Thus,

```
theta= span(0.0, 2*pi, 100)
```

defines a variable `theta` consisting of 100 evenly spaced values starting with `0.0` and ending with `2*pi`.

Now that `theta` has been defined as a list of 100 numbers, any function of `theta` has a concrete representation as a list of 100 numbers — namely the values of the function at the 100 particular values of `theta`. Hence, variables `x` and `y` representing coordinates of the unit circle are defined with:

```
x= cos(theta);   y= sin(theta)
```

Here, `cos` and `sin` are built-in Yorick functions. Like most Yorick functions, they operate on an entire array of numbers, returning an array of like shape. Hence both `x` and `y` are now lists of 100 numbers – the cosines and sines of the 100 numbers `theta`.

The semicolon marks the end of a Yorick statement, allowing several statements to share a single line. The end of a line (i.e.- a newline) can also mark the end of a Yorick statement. However, if any parentheses are open, or if a binary operator or a comma is the last token on the line, then the newline is treated like a space or a tab character and does not terminate the Yorick statement.

If a line ends with backslash, the following newline will never terminate the Yorick statement. (That is, backslash is the continuation character in Yorick.) I recommend that you never use a backslash – end the line to be continued with a binary operator, or leave the comma separating subroutine arguments at the end of the line, or split a parenthetic expression across the line, and it will be continued automatically.

2.1 Creating Arrays

Including the `span` function introduced in the previous section, there are five common ways to originate arrays — that is, to make an array out of scalar values:

`[val1, val2, val3, ...]`
: The most primitive way to build an array is to enumerate the successive elements of the array in square brackets. The length of the array is the number of values in the list. An exceptional case is `[]`, which is called "nil". A Yorick variable has the value nil before it has been defined, and `[]` is a way to write this special value.

`array(value, dimlist)`
: Use `array` to "broadcast" (see Section 2.6 [Broadcasting], page 45) a value into an array, increasing its number of dimensions. A `dimlist` is a standard list of arguments used by several Yorick functions (see Section 2.7 [Dimension Lists], page 47). Often, the pseudo-index range function (see Section 2.3.5 [Pseudo-Index], page 35) is a clearer alternative to the `array` function.

`span(start, stop, number)`
: Use `span` to generate a list of equally spaced values.

`indgen(n)`
`indgen(lower:upper)`
`indgen(start:stop:step)`
: Use `indgen` ("index generator") instead of `span` to generate equally spaced integer values. By default, the list starts with 1.

`spanl(start, stop, number)`
: Use `spanl` ("span logarithmically") to generate a list of numbers which increase or decrease by a constant ratio.

2.2 Interpolating

As I have said, a Yorick array often represents the values of a continuous function at a number of discrete points. In order to find the values of the function at other points, you need to know how it varies between (or beyond) the given points. In general, to interpolate (or extrapolate), you need a detailed understanding of how the function was discretized in the first place. However, when the list of values accurately represents the function, linear interpolation between the known points will suffice. A function which is linear between successive points is called “piecewise linear”.

The `interp` function is a mechanism for converting a list of function values at discrete points into a piecewise linear function which can be evaluated at any point.

```
theta= span(0, pi, 100);
x_circle= cos(theta);
y_circle= sin(theta);
x= span(-2, 2, 64);
y= interp(y_circle, x_circle, x);
```

This code fragment produces a `y` array with the same number of points as `x` (64), with the values of the piecewise linear function defined by the points `(x_circle, y_circle)`. Outside the range covered by `x_circle`, the piecewise linear function remains constant – the simplest possible extrapolation rule.

Regarded as a function of its third argument, `interp` behaves just like the `sin` or `cos` function – its first two arguments are really parameters specifying which piecewise linear function `interp` will evaluate.

The `integ` function works just like `interp`, except that it returns the integral of the piecewise linear function. The integration constant is chosen so that `integ` returns zero at the first point of the piecewise linear function. (This point will actually have the maximum value of x if the x array is decreasing.) Thus, the integral of the piecewise linear approximation to the semicircle and the exact integral of the semicircle can be computed by:

```
yi= integ(y_circle, x_circle, x);
yi_exact= 0.5*(acos(max(min(x,1),-1)) - x*sqrt(1-min(x^2,1)));
```

Again, the piecewise linear function is assumed to remain constant beyond the first and last points specified. Hence, `integ` is a linear function when extrapolating, and piecewise parabolic when interpolating.

Use `integ` only when you need the indefinite integral of a piecewise linear function. Yorick has more efficient ways to compute definite integrals. Again, think of `integ`, like `interp`, as a continuous function of its third argument; the first two arguments are parameters specifying which function.

Neither `interp` nor `integ` makes sense unless its second argument is either increasing or decreasing. There is no way to decide which branch of a multi-valued function should be returned.

Internally, both `interp` and `integ` need a lookup function – that is, a function which finds the index of the point in x_circle just beyond each of the x values. This lookup function can also be called directly; its name is `digitize`.

2.3 Indexing

Yorick has a bewildering variety of different ways to refer to individual array elements or subsets of array elements. In order to master the language, you must learn to use them all. Nearly all of the examples later in this manual use one or more of these indexing techniques, so trust me to show you how to use them later:

- A scalar integer index refers to a specific array element. In a multi-dimensional array, this "element" is itself an array – the specified row, column, or hyperplane.
- An index which is nil or omitted refers to the entire range of the corresponding array dimension.
- An index range of the form `start:stop` or `start:stop:step` refers to an evenly spaced subset of the corresponding array dimension.
- An array of integers refers to an arbitrary subset, possibly including repeated values, of the corresponding array dimension.
- The special symbol `-`, or `-:start:stop`, is the pseudo-index. It inserts an index in the result which was not present in the array being indexed. Although this sounds recondite, the pseudo-index is often necessary in order to make two arrays conformable for a binary operation.
- The special symbol `..` or `*` is a rubber-index. A rubber-index stands for zero or more dimensions of the array being indexed. This syntax allows you to specify a value for the final index of an array, even when you don't know how many dimensions the array has. The `..` version leaves the original dimension count intact; the `*` version collapses all dimensions into a single long dimension in the result.
- The special symbol `+` marks an index for matrix multiplication.
- Many statistical functions, such as `sum`, `avg`, and `max` can be applied along a specific dimension of an array. These reduce the rank of the resulting array, just like a scalar integer index value.
- Several finite difference functions, such as `zcen` (zone center) and `cum` (cumulative sums) can be applied along a specific array dimension. You can use these to construct numerical analogues to the operations of differential and integral calculus.

2.3.1 Scalar indices and array order

An array of objects is stored in consecutive locations in memory (where each location is big enough to hold one of the objects). An array `x` of three numbers is stored in the order `[x(1), x(2), x(3)]` in three consecutive slots in memory. A three-by-two array `y` means nothing more than an array of two arrays of three numbers each. Thus, the six numbers are stored in two contiguous blocks of three numbers each: `[[x(1,1), x(2,1), x(3,1)], [x(1,2), x(2,2), x(3,2)]]`.

A multi-dimensional array may be referenced using fewer indices than its number of dimensions. Hence, in the previous example, `x(5)` is the same as `x(2,2)`, since the latter element is stored fifth.

Although most of Yorick's syntax follows the C language, array indexing is designed to resemble FORTRAN array indexing. In Yorick, as in FORTRAN, the first (leftmost) dimension of an array is always the index which varies fastest in memory. Furthermore, the first element along any dimension is at index 1, so that a dimension of length three can be referenced by index 1 (the first element), index 2 (the second element), or index 3 (the third element).

If this inconsistency bothers you, here is why Yorick indexing is like FORTRAN indexing: In C, an array of three numbers, for example, is a data type on the same footing as the data type of each of its three members; by this trick C sidesteps the issue of multi-dimensional arrays — they are singly arrays of objects of an array data type. While this picture accurately reflects the way the multi-dimensional array is stored in memory, it does not reflect the way a multi-dimensional array is used in a scientific computer program.

In such a program, the fact that the array is stored with one or the other index varying fastest is irrelevent — you are equally likely to want to consider as a "data type" a slice at a constant value of the first dimension as of the second. Furthermore, the length of every dimension varies as you vary the resolution of the calculation in the corresponding physical direction.

2.3.2 Selecting a range of indices

You can refer to several consecutive array elements by an index range: `x(3:6)` means the four element subarray `[x(3), x(4), x(5), x(6)]`.

Occasionally, you also want to refer to a sparse subset of an array; you can add an increment to an index range by means of a second colon: `x(3:7:2)` means the three element subarray `[x(3), x(5), x(7)]`.

A negative increment reverses the order of the elements: `x(7:3:-2)` represents the same three elements, but in the opposite order `[x(7), x(5), x(3)]`. The second element mentioned in the index range may not actually be present in the resulting subset, for example, `x(7:2:-2)` is the same as `x(7:3:-2)`, and `x(3:6:2)` represents the two element subarray `[x(3), x(5)]`.

Just as the increment defaults to 1 if it is omitted, the start and stop elements of an index range also have default values, namely the first and last possible index values. Hence, if `x` is a one-dimensional array with 10 elements, `x(8:)` is the same as `x(8:10)`. With a negative increment, the defaults are reversed, so that `x(:8:-1)` is the same as `x(10:8:-1)`.

A useful special case of the index range default rules is `x(::-1)`, which represents the array `x` in reverse order.

Beware of a minor subtlety: `x(3:3)` is not the same thing as `x(3)`. An index range always represents an array of values, while a scalar index represents a single value. Hence, `x(3:3)` is an array with a single element, `[x(3)]`.

2.3.3 Nil index refers to an entire dimension

When you index a multi-dimensional array, very often you want to let one or more dimensions be "spectators". In Yorick, you accomplish this by leaving the corresponding index blank:

```
x(3,)
x(,5:7)
y(,::-1,)
x(,)
```

In these examples, `x` is a 2-D array, and `y` is a 3-D array. The first example, `x(3,)`, represents the 1-D array of all the elements of `x` with first index 3. The second represents a 2-D array of all of the elements of `x` whose second indices are 5, 6, or 7. In the third example, `y(,::-1,)` is the `y` array with the elements in reverse order along its middle index. The fourth expression, `x(,)`, means the entire 2-D array `x`, unchanged.

2.3.4 Selecting an arbitrary list of indices

An index list is an array of index values. Use an index list to specify an arbitrary subset of an array: `x([5,1,2,1])` means the four element array `[x(5), x(1), x(2), x(1)]`. The `where` function returns an index list:

```
list= where(x > 3.5)
y= x(list)
```

These lines define the array `y` to be the subset of the `x` array, consisting of the elements greater than 3.5.

Like the result of an index range, the result of an index list is itself an array. However, the index list follows a more general rule: The dimensionality of the result is the same as the dimensionality of the index list. Hence, `x([[5, 1], [2, 1]])` refers to the two dimensional array `[[x(5), x(1)], [x(2), x(1)]]`. The general rule for index lists is:

> Dimensions from the dimensions of the index list; values from the array being indexed.

Note that the scalar index value is a special case of an index list according to this rule.

The rule applies to multi-dimensional arrays as well: If `x` is a five-by-nine array, then `x(, [[5, 1], [2, 1]])` is a five-by-two-by-two array. And `x([[5, 1], [2, 1]], 3:6)` is a two-by-two-by-four array.

2.3.5 Creating a pseudo-index

A binary operation applied between two arrays of numbers yields an array of results in Yorick – the operation is performed once for each corresponding pair of elements of the operands. Hence, if `rho` and `vol` are each four-by-three arrays, the expression `rho*vol` will be a four-by-three array of products, starting with `rho(1,1)*vol(1,1)`.

This extension of binary operations to array operands is not always appropriate. Instead of operating on the corresponding elements of arrays of the same shape, you may want to perform the operation between all pairs of elements. The most common example is the outer product of two vectors `x` and `y`:

```
outer= x*y(-,);
```

Here, if `x` were a four element vector, and `y` were a three element vector, `outer` would be the four-by-three array `[[x(1)*y(1), x(2)*y(1), x(3)*y(1), x(4)*y(1)], [x(1)*y(2), x(2)*y(2), x(3)*y(2), x(4)*y(2)], [x(1)*y(3), x(2)*y(3), x(3)*y(3), x(4)*y(3)]]`.

In Yorick, this type of multiplication is still commutative. That is, `x*y(-,)` is the same as `y(-,)*x`. To produce the three-by-four transpose of the array `outer`, you would write `x(-,)*y`.

I call the `-` sign, when used as an index, a pseudo-index because it actually inserts an additional dimension into the result array which was not present in the array being indexed. By itself, the expression `y(-,)` is a one-by-three array (with the same three values as the three element vector `y`). You may insert as many pseudo-indices into a list of subscripts as you like, at any location you like relative to the actual dimensions of the array you are indexing. Hence, `outer(-,-,,-,)` would be a one-by-one-by-four-by-one-by-three array.

By default, a pseudo-index produces a result dimension of length one. However, by appending an index range to the `-` sign, separated by a colon, you can produce a new dimension of any convenient length:

```
x= span(-10, 10, 100)(,-:1:50);
y= span(-5, 5, 50)(-:1:100,);
gauss2d= exp(-0.5*(x^2+y^2))/(2.0*pi);
```

computes a normalized 2-D Gaussian function on a 100-by-50 rectangular grid of points in the xy-plane. The pseudo-index `-:1:50` has 50 elements, and `-:1:100` has 100. For a pseudo-index of non-unit length, the values along the actual dimensions are simply copied, so that `span(-10, 10, 100)(,-:1:50)` is a 100-by-50 array consisting of 50 copies of `span(-10, 10, 100)`.

If only the result `gauss2d` were required, a single default pseudo-index would have sufficed:

```
gauss2d= exp(-0.5*( span(-10,10,100)^2 +
                    span(-5,5,50)(-,)^2 )) / (2.0*pi);
```

However, the rectangular grid of points (`x`,`y`) is often required – as input to plotting routines for example.

2.3.6 Numbering a dimension from its last element

Array index values are subtly asymmetric: An index of 1 represents the first element, 2 represents the second element, 3 the third, and so on. In order to refer to the last, or next to last, or any element relative to the final element, you apparently need to find out the length of the dimension.

In order to remedy this asymmetry, Yorick interprets numbers less than 1 relative to the final element of an array. Hence, `x(1)` and `x(2)` are the first and second elements of `x`, while `x(0)` and `x(-1)` are the last and next to last elements, and so on.

With this convention for negative indices, many Yorick programs can be written without the need to determine the length of a dimension:

```
deriv= (f(3:0)-f(1:-2)) / (x(3:0)-x(1:-2));
```

computes a point-centered estimate of the derivative of a function `f` with values known at points `x`. (A better way to compute this derivative is to use the `pcen` and `dif` range functions described below. See Section 2.3.10 [Rank preserving (finite difference) range functions], page 41.)

In this example, the extra effort required to compute the array length would be slight:

```
n= numberof(f);
deriv= (f(3:n)-f(1:n-2)) / (x(3:n)-x(1:n-2));
```

However, using the negative index convention produces faster code, and generalizes to multi-dimensional cases in an obvious way.

The negative index convention works for scalar index values and for the start or stop field of an index range (as in the example). Dealing with negative indices in an index list would slow the code down too much, so the values in an index list may not be zero or negative.

2.3.7 Using a rubber index

Many Yorick functions must work on arrays with an unknown number of dimensions. Consider a filter response function, which takes as input a spectrum (brightness as a function of photon energy), and returns the response of a detector. Such a function could be passed a spectrum and return a scalar value. Or, it might be passed a two dimensional array of spectra for each of a list of rays, and be expected to return the corresponding list of responses. Or a three dimensional array of spectra at each pixel of a two dimensional image, returning a two dimensional array of response values.

In order to write such a function, you need a way to say, "and all other indices this array might have". Yorick's rubber-index, `..`, stands for zero or more actual indices of the array being indexed. Any indices preceding a rubber-index are "left justified", and any following it are "right justified". Using this syntax, you can easily index an array, as long as you know that the dimension (or dimensions) you are interested in will always be first, or last – even if you don't know how many spectator dimensions might be present in addition to the one your routine processes.

Thus, as long as the spectral dimension is always the final dimension of the input `brightness` array,

```
brightness(..,i)
```

will always place the `i` index in the spectral dimension, whether `brightness` itself is a 1-D, 2-D, or 3-D array.

Similarly, `x(i,..)` selects a value of the first index of `x`, leaving intact all following dimensions, if any. Constructions such as `x(i,j,..,k,l)` are also legal, albeit rarely necessary.

A second form of rubber-index collapses zero or more dimensions into a single index. The length of the collapsed dimension will be the product of the lengths of all of the dimensions it replaces (or 1, if it replaces zero dimensions). The symbol for this type of rubber index is an asterisk `*`. For example, if `x` were a five-by-three-by-four-by-two array, then `x(*)` would be a 1-D array of 120 elements, while `x(,*,)` would be a five-by-twelve-by-two array.

If the last actual index in a subscripted array is nil, and if this index does not correspond to the final actual dimension of the array, Yorick will append a `..` rubber-index to the end of the subscript list. Hence, in the previous example, `x()` is equivalent to `x(,..)`, which is equivalent to `x(..)`, which is equivalent to simply `x`. This rule is the only rogue in Yorick's array subscripting stable, and I am mightily tempted to remove it on grounds of linguistic purity. When you mean, "and any other dimensions which might be present," use the `..` rubber-index, not a nil index. Use a trailing nil index only when you mean, "and the single remaining dimension (which I know is present)."

2.3.8 Marking an index for matrix multiplication

Yorick has a special syntax for matrix multiplication, or, more generally, inner product:

```
A(,+)*B(+,)
B(+,)*A(,+)
x(+)*y(+)
P(,+,,)*Q(,,+)
```

In the first example, `A` would be an L-by-M array, `B` would be an M-by-N array, and the result would be the L-by-N matrix product. In the second example, the result would be the N-by-L transpose of the first result. The general rule is that all of the "spectator" dimensions of the left operand precede the spectator dimensions of the right operand in the result.

The third example shows how to form the inner product of two vectors `x` and `y` of equal length. The fourth example shows how to contract the second dimension of a 4-D array `P` with the third dimension of the 3-D array `Q`. If `P` were 2-by-3-by-4-by-5 and `Q` were 6-by-7-by-3, the result array would be 2-by-4-by-5-by-6-by-7.

Unlike all of the other special subscript symbols (nil, `-`, `..`, and `*` so far), the `+` sign marking an index for use in an inner product is actually treated specially by the Yorick parser. The `+` subscript is a parse error unless the array (or expression) being subscripted is the left or right operand of a binary `*` operator, which is then parsed as matrix multiplication instead of Yorick's usual element-by-element multiplication. A parse error will also result if only one of the operands has a dimension marked by `+`. Both operands must have exactly one marked dimension, and the marked dimensions must turn out to be of equal length at run time.

2.3.9 Rank reducing (statistical) range functions

The beauty of Yorick's matrix multiplication syntax is that you "point" to the dimension which is to be contracted by placing the `+` marker in the corresponding subscript. In this section and the following section, I introduce Yorick's range functions, which share the "this dimension right here" syntax with matrix multiplication. The topic in this section is the statistical range functions. These functions reduce the rank of an array, as if they were a simple scalar index, but instead of selecting a particular element along the dimension, a statistical range function selects a value based on an examination of all of the elements along the selected dimension. The statistical function is repeated separately for each value of any spectator dimensions. The available functions are:

`min`
`max` returns the minimum (or maximum) value. These are also available as ordinary functions.

`sum` returns the sum of all the values. This is also available as an ordinary function.

`avg` returns the arithmetic mean of all the values. The result will be a real number, even if the input is an integer array. This is also available as an ordinary function.

`ptp` returns the peak-to-peak difference (difference between the maximum and the minimum) among all the values. The result will be positive if the maximum occurs at a larger index than the minimum, otherwise the result will be negative.

`rms` returns the root mean square deviation from the arithmetic mean of the values. The result will be a real number, even if the input is an integer array.

`mnx`

`mxx` returns the index of the element with the smallest (or largest) value. The result is always an integer index value, independent of the data type of the array being subscripted. If more than one element reaches the extreme value, the result will be the smallest index.

The `min`, `max`, `sum`, and `avg` functions may also be applied using ordinary function syntax, which is preferred if you want the function to be applied across all the dimensions of an array to yield a single scalar result.

Given the `brightness` array representing the spectrum incident on a detector or set of detectors, the `mxx` function can be used to find the photon energy at which the incident light is brightest. Assume that the final dimension of `brightness` is always the spectral dimension, and that the 1-D array `gav` of photon energies (with the same length as the final dimension of `brightness`) is also given:

```
max_index_list= brightness(.., mxx);
gav_at_max= gav(max_index_list);
```

Note that `gav_at_max` would be a scalar if `brightness` were a 1-D spectrum for a single detector, a 2-D array if `brightness` were a 3-D array of spectra for each point of an image, and so on.

An arbitrary index range (`start:stop` or `start:stop:step`) may be specified for any range function, by separating the function name from the range by another colon. For example, to select only a relative maximum of `brightness` for photon energies above 1.0, ignoring possible larger values at smaller energies, you could use:

```
i= min(where(gav > 1.0));
max_index_list= brightness(.., mxx:i:0);
gav_at_max= gav(max_index_list);
```

Note the use of `min` invoked as an ordinary function in the first line of this example. (Recall that `where` returns a list of indices where some conditional expression is true.) In the second line, `mxx:i:0` is equivalent to `mxx:i:`. Because of the details of Yorick's current implementation, the former executes slightly faster.

More than one range function may appear in a single subscript list. If so, they are computed from left to right. In order to execute them in another order, you must explicitly subscript the expression resulting from the first application:

```
x= [[1, 3, 2], [8, 0, 9]];
max_min= x(max, min);
min_max= x(, min)(max);
```

The value of `max_min` is 3; the value of `min_max` is 2.

2.3.10 Rank preserving (finite difference) range functions

Because Yorick arrays almost invariably represent function values, Yorick provides numercal equivalents to the common operations of differential and integral calculus. In order to handle functions of several variables in a straightforward manner, these operators are implemented as range functions. Unlike the statistical range functions, which return a scalar result, the finite difference range functions do not reduce the rank of the subscripted array. Instead, they preserve rank, in the same way that a simple index range `start:stop` preserves rank. The available finite difference functions are:

`cum` returns the cumulative sums (sum of all values at smaller index, the first index of the result having a value of 0). The result dimension has a length one greater than the dimension in the subscripted array.

`psum` returns the partial sum (sum of all values at equal or smaller index) up to each element. The result dimension has the same length as the dimension in the subscripted array. This is the same as `cum`, except that the leading 0 value in the result is omitted.

`dif` returns the pairwise difference between successive elements, positive when the function values increase along the dimension, negative when they decrease. The result dimension has a length one less than the dimension of the subscripted array. The `dif` function is the inverse of the `cum` function.

`zcen` returns the pairwise average between successive elements. The result dimension has a length one less than the dimension of the subscripted array. The name is short for "zone center".

`pcen` returns the pairwise average between successive elements. However, the two endpoints are copied unchanged, so that the result dimension has a length one greater than the dimension of the subscripted array. The name is short for "point center".

`uncp` returns the inverse of the `pcen` operation. (Gibberish will be the result of applying this function to an array dimension which was not produced by the `pcen` operation.) The result dimension has a length one less than the dimension of the subscripted array. The name is short for "uncenter point centered".

The derivative *dy/dx* of a function *y(x)*, where *y* and *x* are represented by 1-D arrays `y` and `x` of equal length is:

```
deriv= y(dif)/x(dif);
```

Note that `deriv` has one fewer element than either `y` or `x`. The derivative is computed as if *y(x)* were the piecewise linear function passing through the given points `(x,y)`; there is one fewer line segment (slope) than point.

The values `x` and `y` can be called "point-centered", while the values `deriv` can be called "zone-centered". The `zcen` and `pcen` functions provide a simple mechanism for moving back and forth between point-centered and zone-centered quantities. Usually, there will be several reasonable ways to point-center zone centered data and vice-versa. For example:

```
deriv_pc1= deriv(pcen);
deriv_pc2= y(dif)(pcen)/x(dif)(pcen);
deriv_pc3= y(pcen)(dif)/x(pcen)(dif);
```

For a well-resolved function, the differences among these three arrays will be negligible. That is, the differences are second order in `x(dif)`, which is often the order of the errors in the calculation that produced `x` and `y` in the first place. If `x` and `y` represent *y(x)* more accurately that this, then you must know a better model of the shape of *y(x)* than the simple piecewise linear model, and you should use that model to select `deriv_pc1`, `deriv_pc2`, and `deriv_pc3`, or some other expression.

An indefinite integral may be estimated using the trapezoidal rule:

```
indef_integ= (y(zcen)*x(dif))(psum);
```

Once again, `indef_integ` has one fewer point than `x` or `y`, because there is one fewer trapezoid than point. This time, however, `indef_integ` is not zone centered. Instead, `indef_integ` represents values at the upper (or lower) boundaries `x(2:)` (or `x(:-1)`). Often, you want to think of the integral of *y(x)* as a point centered array of definite integrals from `x(1)` up to each of the `x(i)`. In this case (which actually arises more frequently), use the `cum` function instead of `psum` in order to produce a result `def_integs` with the same number of points as the `x` and `y` arrays:

```
def_integs= (y(zcen)*x(dif))(cum);
```

For single definite integrals, the matrix multiply syntax can be used in conjunction with the `dif` range function. For example, suppose you know the transmission fraction of the filter, `ff`, at several photon energies `ef`. That is, `ff` and `ef` are 1-D arrays of equal length, specifying a filter transmission function as the piecewise linear function connecting the given points. The final dimension of an array `brightness` represents an incident spectrum (any other dimensions represent different rays, say one ray per pixel of an imaging detector). The 1-D array `gb` represents the group boundary energies – that is, the photon energies at the boundaries of the spectral groups represented in `brightness`. The following Yorick statements compute the detector response:

```
filter= integ(ff, ef, gb)(dif);
response= brightness(..,+)*filter(+);
```

For this application, the correct interpolation routine is `integ`. The integrated transmission function is evaluated at the boundaries `gb` of the groups; the effective transmission fraction for each group is the difference between the integral at the upper bin boundary and the lower bin boundary. The `dif` range function computes these pairwise differences. Since the `gb` array naturally has one more element than the final dimension of `brightness` (there is one more group boundary energy than group), and since `dif` reduces the length of a dimension by one, the `filter` array has the same length as the final dimension of `brightness`.

Note that the `cum` function cannot be used to integrate `ff(ef)`, because the points `gb` at which the integral must be evaluated are not the same as the points `ef` at which the integrand is known. Whenever the points at which the integral is required are the same (or a subset of) the points at which the integrand is known, you should perform the integral using the `zcen`, `dif`, and `cum` index functions, instead of the more general `integ` interpolator.

2.4 Sorting

The function `sort` returns an index list:

```
list= sort(x);
```

The list has the same length as the input vector `x`, and list(1) is the index of the smallest element of `x`, list(2) the index of the next smallest, and so on. Thus, `x(list)` will be `x` sorted into ascending order.

By returning an index list instead of the sorted array, `sort` simplifies the co-sorting of other arrays. For example, consider a series of elaborate experiments. On each experiment, thermonuclear yield and laser input energy are measured, as well as fuel mass. These might reasonably be stored in three vectors `yield`, `input`, and `mass`, each of which is a 1D array with as many elements as experiments performed. The order of the elements in the arrays maight naturally be the order in which the experiments were performed, so that `yield(3)`, `input(3)`, and `mass(3)` represent the third experiment. The experiments can be sorted into order of increasing gain (yield per unit input energy) as follows:

```
list= sort(yield/input);
yield= yield(list);
input= input(list);
mass= mass(list);
```

The inverse list, which will return them to their orginal order, is:

```
invlist= list;  /* faster than  array(0, dimsof(list)) */
invlist(list)= indgen(numberof(list));
```

The `sort` function actually sorts along only one index of a multi-dimensional array. The dimensions of the returned list are the same as the dimensions of the input array; the values are indices relative to the beginning of the entire array, not indices for the dimension being sorted. Thus, `x(sort(x))` is the array `x` sorted so that the elements along its first dimension are in ascending order. In order to sort along another dimension, pass `sort` a second argument – `x(sort(x,2))` will be `x` sorted into increasing order along its second dimension, `x(sort(x,3))` along its third dimension, and so on.

A related function is `median`. It takes one or two arguments, just like `sort`, but its result – which is, of course, the median values along the dimension being sorted – has one fewer dimension than its input.

2.5 Transposing

In a carefully designed Yorick program, array dimensions usually wind up where you need them. However, you may occasionally wind up with a five-by-seven array, instead of the seven-by-five array you wanted. Yorick has a very general `transpose` function:

```
x623451= transpose(x123456);
x561234= transpose(x123456, 3);
x345612= transpose(x123456, 5);
x153426= transpose(x123456, [2,5]);
x145326= transpose(x123456, [2,5,3,4]);
```

```
x653124= transpose(x123456, [2,5], [1,4,6]);
```

Here, `x123456` represents a six-dimensional array (hopefully these will be rare). The same array with its first and last dimensions transposed is called `x623451`; this is the default result of `transpose`. The `transpose` function can take any number of additional arguments to describe an arbitrary permutation of the indices of its first argument – the array to be transposed.

A scalar integer value, as in the second and third lines, represents a cyclic permutaion of all the dimensions; the first dimension of the input becomes the Nth dimension of the result, for an argument value of N.

An array of integer values represents a cyclic permutation of the specified dimensions. Hence, in the fifth example, `[2,5,3,4]` means that the second dimension becomes the fifth, the fifth becomes the third, the third becomes the fourth, and the fourth becomes the second. An arbitrary permutation may be built up of a number of cyclic permutations, as shown in the sixth example.

One additional problem can arise in Yorick – you may not know how many dimensions the array to be transposed has. In order to deal with this possibility, either a scalar argument or any of the numbers in a cyclic permutation list may be zero or negative to count from the last dimension. That is, 0 represents the last dimension, -1 the next to last, -2 the one before that, and so on.

Typical transposing tasks are: (1) Move the first dimension to be last, and all the others back one cyclically. (2) Move the last dimension first, and all the others forward one cyclically. (3) Transpose the first two dimensions, leaving all others fixed. (4) Transpose the final two dimensions, leaving all others fixed. These would be accomplished, in order, by the following four lines:

```
x234561= transpose(x123456, 0);
x612345= transpose(x123456, 2);
x213456= transpose(x123456, [1,2]);
x123465= transpose(x123456, [0,-1]);
```

2.6 Broadcasting and conformability

Two arrays need not have identical shape in order for a binary operation between them to make perfect sense:

```
y= a*x^3 + b*x^2 + c*x + d;
```

The obvious intent of this assignment statement is that `y` should have the same shape as `x`, each value of `y` being the value of the polynomial at the corresponding `y`.

Alternatively, array valued coefficients `a`, `b`, ..., represent an array of several polynomials – perhaps Legendre polynomials. Then, if `x` is a scalar value, the meaning is again obvious; `y` should have the same shape as the coefficient arrays, each `y` being the value of the corresponding polynomial.

A binary operation is performed once for each element, so that `a+b`, say, means

```
a(i,j,k,l) + b(i,j,k,l)
```

for every `i, j, k, l` (taking `a` and `b` to be four dimensional). The lengths of corresponding indices must match in order for this procedure to make sense; Yorick signals a "conformability error" when the shapes of binary operands do not match.

However, if `a` had only three dimensions, `a+b` still makes sense as:

```
a(i,j,k) + b(i,j,k,l)
```

This sense extends to two dimensional, one dimensional, and finally to scalar `a`:

```
a + b(i,j,k,l)
```

which is how Yorick interpreted the monomial `x^3` in the first example of this section. The shapes of `a` and `b` are conformable as long as the dimensions which they have in common all have the same lengths; the shorter operand simply repeats its values for every index of a dimension it doesn't have. This repitition is called "broadcasting".

Broadcasting is the key to Yorick's array syntax. In practical situations, it is just as likely for the `a` array to be missing the second (`j`) dimension of the `b` array as its last (`l`) dimension. To handle this case, Yorick will broadcast any unit-length dimension in addition to a missing final dimension. Hence, if the `a` array has a second dimension of length one, `a+b` means:

```
a(i,1,k,l) + b(i,j,k,l)
```

for every `i, j, k, l`. The pseudo-index can be used to generate such unit length indices when necessary (see Section 2.3.5 [Pseudo-Index], page 35).

2.7 Dimension Lists

You should strive to write Yorick programs in such a way that you never need to refer to the lengths of array dimensions. Array dimensions are usually of no direct significance in a calculation; your programs will tend to be clearer if only the arrays themselves appear.

In practice, unfortunately, you can't always get by without mentioning dimension lengths. Two functions are important:

`numberof(x)`
: returns the total number of items in the array `x`.

`dimsof(x)`
: returns a list consisting of the number of dimensions of the array `x`, followed by the length of each of those dimensions. Thus, if `x` were a nine-by-two-by-six array, `dimsof(x)` would return `[3,9,2,6]`, while if `x` were a scalar value, `dimsof(x)` would return `[0]`.

`dimsof(x, y, z, ...)`
: With multiple arguments, the `dimsof` function returns the dimension list of the array `x+y+z+...`, or `[]` if the input arrays are not conformable.

The `array` function (see Section 2.1 [Creating Arrays], page 30), and the more arcane functions `add_variable`, `add_member`, and `reshape` all have parameter lists ending with one or more "dimension list" parameters. Each parameter in a dimension list can be either a scalar integer value, representing the length of a single dimension, or a list of integers in the format returned by `dimsof` to represent zero or more dimensions. Several arguments can be used to build up a complicated dimension list:

```
x1= array(0.0, 9, 2, 6);      /* 9-by-2-by-6 array of 0.0 */
x2= array(0.0, [3,9,2,6]);  /* another 9-by-2-by-6 array */
x3= array(0.0, 9, [0], [2,2,6]);   /* ...and yet another */

flux= array(0.0, 3, dimsof(z), numberof(groups));
```

In the final example, the `flux` might represent the three components of a flux vector, at each of a number of positions `z`, and for each of a number of photon energies `groups`. The first dimension of `flux` has length three, corresponding to the three components of each flux vector. The last dimension has the same length as the `groups` array. In between are zero or more dimensions – whatever the dimensions of the array of positions `z`.

By using the rubber index syntax (see Section 2.3.7 [Rubber-Index], page 37), you can extract meaningful slices of the `flux` array without ever needing to know how many dimensions `z` had, let alone their lengths.

3 Embedding Compiled Routines Inside Yorick

(Very preliminary notes. Not heavily tested either. See the distribution Drat and MathC, and Math directories for examples.)

You can create a custom version of Yorick containing your own C or Fortran compiled code. If you are careful, your custom version will be easily portable to any site where Yorick has been installed. You will considerably ease portability problems (read "future hassles for yourself") by writing in C, which is a portable language, as opposed to Fortran, which is not.

If you do choose Fortran, stick to a strict subset of ANSI Fortran 77, in which you do not attempt to pass character variables into or out of interface routines, nor put them in common blocks. Also, try not to use common blocks to pass inputs to or receive outputs from your interface routines.

Whether you write in Fortran or C, *do not* attempt to do I/O of any sort in your compiled code. The whole idea of embedding routines inside Yorick is to let Yorick handle all I/O – text and graphics.

In order to build a custom Yorick, you need to find the Yorick home directory '`Y_HOME`' on the platform where you will be compiling. You may need to ask whoever installed Yorick where this is. If Yorick was installed as recommended, the home directory will be the '`yorhome`' sibling of the '`bin`' directory from which Yorick was launched. The name of Yorick's launch directory is contained in the `Y_LAUNCH` variable; start Yorick on the machine where you will be compiling and type `Y_LAUNCH` to print this name.

Make a directory in which to build your custom version of Yorick, and put your C and/or Fortran source files there. Copy '`Y_HOME/Maketmpl`' into that directory, change its name to "Makefile", and modify its first section to tell make how to compile your source code.

You next need to write a Yorick include file which will be loaded whenever your custom Yorick starts. This file is read not only by Yorick when it starts, but also by the Codger automatic wrapper code generator when you make the custom version. Codger generates a table of all of the compiled objects (functions, global variables, or Fortran common blocks) which can be referenced from the interpreter. It will also generate wrapper code to call your compiled functions, if you decide not to do this yourself. In this include file:

```
extern my_func;
```

```
/* DOCUMENT my_func(input)
     returns the frobnostication of the array INPUT.
 */
```

connects `my_func` to a compiled function `Y_my_func` of type BuiltIn, which you are responsible for writing. See '`Y_HOME/ydata.h`' for the definition of the BuiltIn function type. This function must pop its arguments off of Yorick's interpreter stack using routines declared in '`Y_HOME/ydata.h`', and push its result back onto the stack. If you want codger to generate '`Y_my_func`' for you automatically, do this instead:

```
func my_func(input)
/* DOCUMENT my_func(input)
     returns the frobnostication of the array INPUT.
 */
{
  return my_func_raw(input, numberof(input));
}
extern my_func_raw;
/* PROTOTYPE
   double my_func_C_name(double array input, long length)
 */
```

This generates a wrapper for a C function which takes a single array as input and returns a scalar result. If the function had been Fortran, it would have looked like this (Fortran passes all arguments by reference – that is, as if they were arrays):

```
func my_func(input)
/* DOCUMENT my_func(input)
     returns the frobnostication of the array INPUT.
 */
{
  return my_func_raw(input, numberof(input));
}
extern my_func_raw;
/* PROTOTYPE FORTRAN
   double my_func_Fortran_name(double array input, long array length)
 */
```

Legal data types for the function return result in the `PROTOTYPE` comment are: void (i.e.- a subroutine), char, short, int, long, float, or double.

Legal data types for the function parameters in the `PROTOTYPE` comment are: void (only if there are no other parameters), char, short, int, long, float, double, string (char *, guaranteed 0-

terminated), or pointer (void *). These may be followed by the word "array", which becomes "*" in the C source code, to indicate an array of that type. The parameter name is optional.

The `DOCUMENT` comment should start with `/* DOCUMENT`. They will be returned by the interpreted command `help, my_func`, and be included in the poor- man's document produced by Yorick's "mkdoc" command (see '`Y_HOME/include/mkdoc.i`').

```
extern my_global;
reshape, my_global, datatype;
```

attaches the interpreted variable `my_global` to a C-compiled global of the same name, which has the data type `datatype` (this must have been declared in a previous struct or be one of the primitive types). If you want `my_global` to be attached to a global variable of a different name, use:

```
extern my_global;
/* EXTERNAL my_global_C_name */
reshape, my_global, datatype;
```

To attach to a Fortran common block, say

```
double var1, var2, var3
common /my_common/ var1, var2, var3
save /my_common/
```

(note that this doesn't make sense unless the common block is saved outside the scope of the functions in which it is used) use:

```
struct my_common_type { double var1, var2, var3; }
extern my_common;
/* EXTERNAL FORTRAN my_common */
reshape, my_common, my_common_type;
```

If you mix double, integer, and real data in a single common block, you can ensure that you won't have any alignment difficulties by putting all the doubles first, followed by integers and reals. If you don't do this, you're relying on the existence of a Fortran compiler switch which forces proper data alignment – some machine someday won't have this.

If you've done it right, to port your code to another machine, you just move your source directory, change `Y_HOME` in the Makefile to point to the public Yorick directory on the new machine, type "make Makefile" to stuff the correct compiler options into the second part of the Makefile (the part you didn't write), then type "make" to build your code on the new machine.

Concept Index

(Index is nonexistent)

Table of Contents