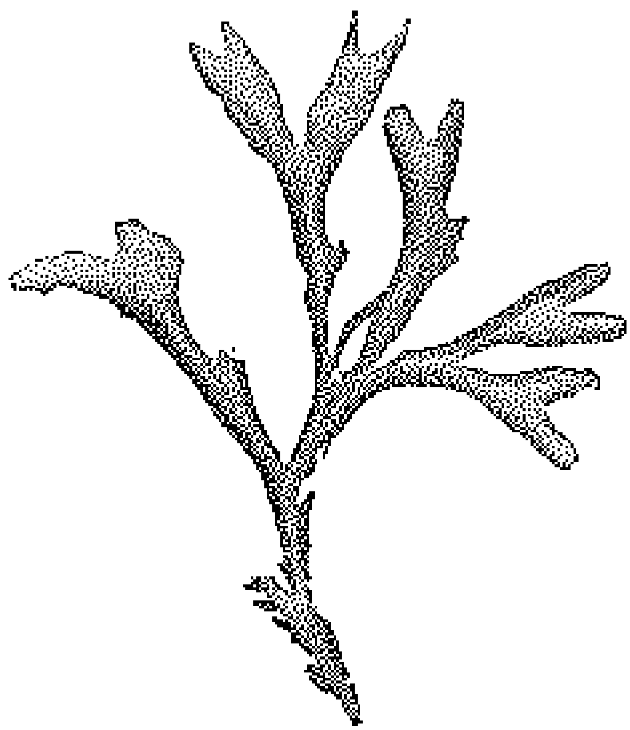
The Algae Programming Language

For algae 4.3.6; February 22, 2004.



by K. Scott Hunziker

Algae 4.3.6; February 22, 2004

Caution: Use of this product as a substitute for a healthy physical, emotional, and spiritual life is not recommended and could be harmful.

1 About Algae

Algae is an interpreted language for numerical analysis. Algae was developed because we needed a fast and versatile tool, capable of handling large problems. Algae has been applied to interesting dynamics problems in aerospace and related fields for more than a decade.

This document describes both Algae, the programming language, and `algae`, the interpreter. Neither one of them is complete. Even so, they offer some unique capabilities that we hope you'll find useful.

Algae was designed and implemented by Scott Hunziker and Mike Brennan.

Algae grew up in the "Structures Outback" of the Boeing Aerospace Company (now the Boeing Defense & Space Group). Way back in 1989, Bill Russell convinced Bob Vos, the head of Boeing's ISM project, that ISM needed an interactive language. With Dave Beste's help, Bill put together a crude system (it looked a lot like MATLAB) with a recursive descent parser. They called it ISMLAB. After Bill got a real job, Scott Hunziker took over and made major changes. ISMLAB, along with the entire ISM project, was cancelled ignominiously in 1991.

Scott Hunziker and Mike Brennan began work on a successor to ISMLAB called AMPL (A Matrix Programming Language) with internal Boeing funding. (The author remembers well the first day of their collaboration, when he inadvertently showed Mike that AMPL could multiply two matrices and get the wrong answer.) As AMPL took shape, the internal funding dried up, leaving Scott to maintain it in his spare time. Boeing generously released AMPL to the public in 1994. The name was changed to Alki in 1995 because, by then, an unrelated package called AMPL was being distributed through netlib. That name also had a conflict, and it was finally changed to Algae in early 1997.

Thanks to Don Morris for contributing the plotting code (not to mention countless bug reports), and Ian Searle for advice, constructive criticism, and pizza.

2 Primer

What follows is a quick introduction to Algae. It shows many simple examples, but leaves out the detailed descriptions. See Chapter 4 [Language], page 17, for more details about the Algae language.

The first thing that you'll need to know is how to get Algae started. If your system is properly set up, it's a simple matter of typing the command '`algae`'. This brings Algae up in interactive mode; a prompt is displayed and it waits for you to start typing statements. Later we'll discuss other options, such as giving Algae a file of statements to execute in "batch" mode.

This manual is available on-line through the `info` function; type `info()` to view it. You can go to a specific topic by naming it as an argument. For example, `info("operators")` takes you directly to the description of Algae's operators.

An Algae *statement* describes the operations to be performed. For example, the statement

```
1+sin(2)
```

tells Algae to add 1 to the sine of 2. Statements may be terminated with a newline, a semicolon, or a question mark; the results are printed unless a semicolon is used.

The statement

```
printf ("hello, world\n");
```

prints a hopeful little "hello, world" message to the terminal. This is exactly like the C language, right down to the '`\n`' escape sequence to indicate a newline. If you know C, you'll probably notice many other similarities between its syntax and that of Algae.

Like most computer languages, Algae has *variables* to which values can be assigned. In Algae, these variables need not be declared before being used, and may be created or destroyed during a session. If you type the statement `x=1`, the variable `x` will be created if it doesn't already exist. If `x` already had a value, then assignment to `x` destroys its previous contents.

The values taken on by variables are known as *entities*, which have various *classes* such as '`scalar`', '`vector`', '`matrix`', '`table`', etc. A builtin function called `class` returns the class of its argument, so if you make the assignment `x=1` then the statement `class(x)` returns '`"scalar"`'.

Notice that a scalar is not the same thing as a one-element vector or as a one-by-one matrix. The builtin functions `scalar`, `vector`, and `matrix` may be used to convert from one to another. For example, `matrix(7)` returns a matrix with one row and one column, its single element having the value 7. Algae will often make these conversions between classes automatically. In the code

```
x = [ 3, 2, 1 ];
y = sort (x);
```

the `sort` function first converts its matrix argument `x` into a vector and then returns another vector with the same elements but sorted in increasing order. (An expression enclosed in brackets defines a matrix, as we'll discuss later.)

Besides its class, an entity may have other attributes which are stored as its *members*. For example, the number of rows in a matrix is stored in a member called "nr". Members are referenced with the "dot" operator, so if `M` is a matrix, then `M.nr` returns its row size. Most entities have one or more predefined members (such as `nr` in matrices) that you cannot directly modify. You can create new members simply by assignment.

A function called `show` prints information about an entity and its members. Another function, `members`, returns a vector containing the names of all the members of its argument.

When a non-existent variable or member is referenced, the special value `NULL` is returned. For example, the line

```
a = 1; a.nr
```

(which consists of two statements) prints "NULL", since scalars do not start life with a member `nr`.

The NULL constant may be used on the right-hand side of assignments, effectively deleting the previous value of an entity. Actually, the entity still exists, but it has the value NULL. You can perform a number of other operations on NULL, such as in

```
if (x != NULL) { x.class? }
```

All array entities (and that includes scalars) have a member called `type`, which may have one of these values: "integer", "real", "complex", or "character". The constant `1` is an integer, but `1.0` has real type. Algae has no complex constant like Fortran does—you have to use an expression such as `sqrt(-1)`. Users often make the assignment `i=sqrt(-1)` when they first start up Algae and then use expressions like `1+2*i` for their complex numbers.

The "character" type refers to a string of characters. It is specified using double-quotes as we did for `"hello world"` above. A character scalar like `"1"` is different than an integer scalar like `1`, and an expression like `1+"1"` is not allowed.

A *vector* is a one-dimensional array of values. For example, `x=1,2,3` specifies a vector with three elements. The elements in a vector are numbered starting at one, and the total number of elements is given by its `ne` member. All of the elements have the same type.

Actually, the comma character is Algae's "append" operator. You can put several expressions together in a vector, as in

```
v = 1, sin(2), 3+4;
```

A "subvector" expression may be used to specify a particular element or elements. You do that simply by following the vector with a *specifier* enclosed in brackets. For example, `v[3]` gives a scalar having the value of the third element of `v`. If the specifier is a scalar, then the result is a scalar; otherwise, the result is a vector.

A more complicated example is

```
aset = 3, 9, 15;
x = v[aset][2];
```

Here, `x` gets the value of the ninth element of `v`. You can also assign to a subvector, so `v[1]=0` sets the first element of `v` to zero.

Vectors have a member `eid` (it stands for *element id*) that contains labels for its elements. You don't need labels (`eid` may be NULL), but they can be pretty useful. For one thing, they can help you to avoid errors by not allowing you to perform certain operations unless the labels match. If you try to add two vectors and they both have labels, then the labels must be identical.

You can also use labels instead of element numbers in a subvector expression. You do this by using character strings as the specifier. For example, the code

```
weight =        172,    216,     188;
weight.eid =  "Tom", "Dick", "Harry";
```

sets up the vector `weight` with character string labels. Then `weight["Dick"]` gives Dick's weight of 216.

Vectors can be generated by using the colon operator. The expression `1:5:2` gives a vector whose first element is `1`, last element is no more than `5`, and has a difference of `2` between each successive element. In other words, `1:5:2` is the same as `1,3,5`. If you leave off the second colon and the third operand, then Algae infers a `1` for the third operand. Thus, if `n=100`, then `1:n` is the vector containing all the integers from `1` through `100`.

A *matrix* is a two-dimensional array of values. The expression `[1,2;3,4;5,6]` specifies a matrix with three rows and two columns—rows are given as vectors and are separated by semicolons.

Submatrix expressions work just like subvector expressions, but with a semicolon to separate the row specifier from the column specifier. The expression `M[3;2,3]` gives a vector containing the elements of `M` in its third row and second and third columns. If both specifiers are scalars, then the result is a scalar. If only one specifier is a scalar, then the result is a vector. Otherwise, the result is a matrix. The members `nr` and `nc` give the number of rows and columns of a matrix.

Matrices have both row and column labels. They're stored in the members `rid` and `cid`, respectively. As with vectors, character strings used as specifiers in submatrix expressions refer to the labels.

A *table* is an entity that simply holds a collection of other entities. For example, the statements

```
x = 1; y = "foo", "bar";
t = { x; y };
```

result in a table `t` that contains the scalar `x` and the vector `y`. Instead, we could write

```
t = { x = 1; y = "foo", "bar" };
```

and get the same table. In the latter case, though, `x` and `y` exist only inside the table.

You can put any class of entity into a table, even another table. The members are referenced with the "dot" operator, just like the other entities. The line `a={u=1;v=2}; a.u+a.v` prints the value 3. You can add two tables (the members of the right-hand table are inserted into the left-hand table) and subtract two tables (the members of the left-hand table having the same name as a member of the right-hand table are removed).

Algae normally executes statements in the order that it receives them, but the control-flow statements `if`, `for`, `while`, and `try` can change that. The code

```
if (x > 0) { y = 1/x; }
```

is an example of an `if` statement. The parentheses are required around the test expression. If that expression is "true", then the statements following it are executed. The `if` statement may also have an `elseif` part, an `else` part, or both, so

```
if (x > 0)
{
    printf ("positive");
elseif (x < 0)
    printf ("negative");
```

```
else
    printf ("zero");
}
```

prints the sign of `x`. (But you should use the `sign` function, instead.)

As long as only scalars are involved, Algae's relational, equality, and logical operators probably won't surprise you. An expression is `1` if it's true and `0` if it's false. The relational operators are `>`, `>=`, `<`, `<=`. The equality operators are `==` and `!=`. The logical operators are `&`, `|`, and `!`. Two additional logical operators, `&&` and `||`, are special; they are described later.

Where these operators might surprise you is when vectors and matrices are involved. Like most Algae operators, they work on an element-by-element basis. For example, if `A` and `B` are both matrices, then the expression `A==B` has several features:

- `A` and `B` must have the same size.
- Their labels must match.
- The expression returns a matrix with the same size as `A` and `B`, every element of which is either `1` or `0` depending on whether the corresponding elements of `A` and `B` are equal.

You can see, then, that `A==B` doesn't give you a simple true or false answer but rather a matrix of answers.

The `if` statement, however, does need a simple true or false in order to decide whether to execute its statements or not. It does this by recognizing certain entities as false—all others are true. The "false" entities are as follows:

- NULL.
- Numeric entities in which every element is zero.
- Character entities in which every element is `""`.
- Vectors and matrices with no elements.
- Tables with no members.

What gets new users into trouble is a statement like

```
if (A == B) { done = 1; }
```

If `A` and `B` are matrices, then `A==B` returns a matrix of ones and zeros. The `if` statement interprets that as "true" if even one of its elements is nonzero. In words, the above statement starts out "If any element of `A` is equal to the corresponding element of `B`, then . . ." The difficulty is that the element-by-element "equality" operation is not the same as a test of the equality of two arrays. If the latter test is what you really want, then you should use the `equal` function instead.

On the other hand, the statement

```
if (A != B) { done = 0; }
```

does work in both senses. The expression `A!=B` returns a matrix that has nonzero elements where the corresponding elements of `A` and `B` are unequal. Thus this expression also serves as a test of the inequality of the two arrays.

The `&&` ("and") and `||` ("or") operators are special in two ways: they don't perform element-by-element like the other operators in this section, and they "short-circuit" by skipping evaluation of the second operand if the result is already established by the first operand.

Each operand of `&&` and `||` is evaluated for "truth" in the same way that the `if` test does. For `&&`, if the first operand evaluates to "false" then the second operand is not evaluated and the result of the operation is 0. For `||`, if the first operand evaluates to "true" then the second operand is not evaluated and the result of the operation is 1.

For example, in the expression

```
x != NULL && x.type == "integer"
```

`x` is first checked to see if it's NULL. If it is, then the first operand of `&&` is 0 and that's also the result of the entire expression. In that case, the member reference `x.type` is never evaluated. This is convenient, since that would otherwise be an error.

An `if` statement such as

```
if (x < tol) { x = tol; }
```

could be written instead as

```
x < tol && (x = tol);
```

The parentheses are required in the second version, since the precedence of `=` is lower than that of `&&`. Although they accomplish the same thing, the first version is recommended; it is easier to read and executes a bit faster.

Algae has two control-flow statements that perform looping: `while` and `for`. The `while` statement executes a set of statements over and over, as long as a given condition is true. For example, the code

```
a=0; b=1;
while (b < 10000)
{
  c = b;
  b = a+b;
  a = c;
}
c?
```

computes and prints the largest Fibonacci number less than 10000. The interpreter checks to make sure that `b<10000`, executes the statements in the `while` block, and then repeats. The first time that the expression `b<10000` evaluates false, the loop terminates.

The `for` statement also causes looping, but in a different way. Assuming that `v` is a numeric vector, the code

```
for (i in 1:v.ne) { v[i] = 1 / v[i]; }
```

inverts each of its members. Inside the parentheses, the keyword `in` separates an identifier on the left and a vector expression on the right. In this example, the vector expression is `1:v.ne` which contains the integers from 1 to the length of `v`. The `for` loop sets `i` equal to the first element, 1, and then executes the statement `v[i]=1.0/v[i];`. Then `i` is set equal to the second element, and the statement is executed again. This cycle repeats until all of the elements of `1:v.ne` are used.

The previous example also illustrates an important topic concerning both `while` loops and `for` loops. Essentially the same results would be obtained with the statement `v=1/v`. This obviously takes less typing and is easier to read. The really important difference, though, is that it is far more efficient. With the `for` loop, all of the operations (assignment, division, etc.) are performed by the interpreter. Although Algae is fast, it can't possibly compete with doing the same job in C, as `v=1/v` is. On my computer, it's about 60 times faster.

Sometimes it's convenient to interrupt the execution of `while` and `for` loops. The `continue` statement causes another iteration of the loop to begin immediately. If we wanted to invert the nonzero elements of a vector, we could write

```
for (i in 1:v.ne)
{
  if (v[i] == 0) { continue; }
  v[i] = 1 / v[i];
}
```

The `break` statement goes even further, exiting the loop altogether. For example, we could have written our Fibonacci routine as

```
a=0; b=1;
while (1)
{
  c = b;
  if ((b = a+b) > 10000) { break; }
  a = c;
}
c?
```

The `continue` and `break` statements affect execution of only the innermost enclosing loop.

It's important to note that `continue` and `break` are statements, not expressions. The code

```
x < 0 && break;
```

is not valid, since the operands of `&&` must be expressions.

The `try` statement is the remaining control-flow statement. It is used to modify Algae's response to something called an *exception*. When an exception occurs, we say that it has been "raised". Many error conditions (dividing by zero, running out of memory, etc.) cause an exception to be raised. You may also raise an exception directly by calling the `exception` function.

When an exception is raised, Algae's default action is to stop execution. If it's running interactively, it returns to the command line prompt; otherwise it exits completely. This action can be modified by using the `try` statement. If an exception is raised within the `try` block, execution continues at the point immediately following that block. For example, the statement

```
try { i += 1; }
```

increments `i` by one if possible. If an error occurs (say `i` was NULL), then Algae just moves on to execute the next line. Although it's probably not a good idea, we could use this instead of `break` in the previous Fibonacci example:

```
a=0; b=1;
try
{
  while (1)
```

```
    {
      c = b;
      if ((b = a+b) > 10000) { exception (); }
      a = c;
    }
  }
  c?
```

There's a real difference between these approaches, though. In the version using the `break` statement, the result is printed only when the correct value is reached. In the `try` version, the result is printed regardless of why the exception was raised—if Algae received an interrupt signal while in the middle of the loop, it would jump out of the `try` block and then print an incorrect value.

The `try` statement has an optional `catch` clause, which is something like the `else` clause of an `if` statement. Any statements that follow `catch` are executed only if an exception is raised while in the preceding part of the `try` block. Also, exceptions that occur within the `catch` block are not handled specially. For example,

```
try
{
  v = 1/v;
catch
  message ("I can't do that, Dave.");
  exception ();
}
```

would print a message if it had trouble performing the inverse and then stop execution. If no exception occurs prior to the `catch` clause, then execution continues following the end of the `try` statement.

The language syntax does not allow a `break` or `continue` statement to be positioned such that it would cause execution to jump out of the `try` block.

Like other computer languages, Algae has *functions*. Some functions (like `sin` and `printf`) are *builtin*, meaning that they are part of the Algae executable file. Others, called *user functions*, are those written in the Algae language. A function is called by giving its name followed by a parenthesized list of *arguments*. The arguments are separated by semicolons, and their values are passed to the function. Since only the values are passed, the function cannot modify the variables that you pass it.

For example, let's assume that you're calling the function `shady`.

```
x = 1;
y = shady (x);
```

The value returned by `shady` is assigned to `y`. You can't tell by looking at it what class of entity (scalar, matrix, etc.) `shady` returns. In fact, that might even change from one call to the next. Rest assured, though, the value of `x` is still 1, no matter what happened in `shady`.

Algae functions are entities—just like scalars and matrices. That means that you can perform operations on them as you do with other entities. For example, the statement

```
my_sin = sin
```

creates a new function called `my_sin` that works just like the original `sin` function. Of course,

```
sin = NULL;
```

gets rid of the `sin` function completely—probably not a very good idea in most cases.

Functions may be defined during an interactive session or simply included from files. As an example, consider writing a function called "findit" that will look through the elements of a vector for a given value, returning the locations where it found it. The following function should do the job:

```
findit = function (s; v)
{
  local (w; i);
  w = vector ();
  for (i in 1:v.ne)
  {
    if (v[i] == s) { w = w, i; }
  }
  return w;
}
```

(The builtin function `find` does this job much more efficiently.) The `local` statement declares its arguments as having local scope. For example, the assignment to `w` would have no effect outside this function. Without the `local` declaration, the assignment would change the value of `w` globally.

The `return` statement causes execution of the function to terminate and passes it's expression as the function's return value. Recursive calls to a function are no problem. For example, we could write a function to compute factorials as

```
fact = function (n)
{
  if (n < 2) { return 1.0; else return n * fact (n-1); }
}
```

This function has one slight problem. (Several, really, if you consider that it does no error checking.) If we later decide to change its name by typing `factorial=fact`, it still calls function `fact` internally. Now if we're really mean-spirited we can change `fact` as in `fact=sin`; now `factorial` gives wrong answers. The way to handle this is to call the function `self` when you make a recursive function call, as in

```
fact = function (n)
{
  if (n < 2) { return 1.0; else return n * self (n-1); }
}
```

The `self` keyword refers to the current function. Besides recursive function calls, it's also useful for keeping data local to a function. For example, consider a function that returns the "shape" of its argument:

```
shape = function (x)
{
  return self.(class(x)) (x);
};

shape.scalar = function (x) { return NULL; };
shape.vector = function (x) { return x.ne; };
```

```
shape.matrix = function (x) { return x.nr, x.nc; };
```

This `shape` function determines the class of its argument and then calls the appropriate member function. (The standard `shape` function additionally provides some error checking.)

3 Example Programs

Following are several examples designed to show off Algae's capabilities as well as to give you a feel for how it is used. The '`examples`' directory in the source distribution of Algae contains some additional code for perusal.

3.1 Temperature Conversion

This Algae program prints a table of Fahrenheit temperatures and their Celsius equivalents:

```
# Print temperature conversions
fahr = sort (0:300:20, 32, 212);
celsius = (5/9)*(fahr-32);
[fahr;celsius]'?
```

Its output looks like this:

```
[          0.000        -17.78 ]
[          20.00        -6.667 ]
[          32.00         0.000 ]
[          40.00         4.444 ]
[          60.00         15.56 ]
[          80.00         26.67 ]
[          100.0         37.78 ]
[          120.0         48.89 ]
[          140.0         60.00 ]
[          160.0         71.11 ]
[          180.0         82.22 ]
[          200.0         93.33 ]
[          212.0         100.0 ]
[          220.0         104.4 ]
[          240.0         115.6 ]
[          260.0         126.7 ]
[          280.0         137.8 ]
[          300.0         148.9 ]
```

This program does quite a bit in only four lines. The entire first line is a *comment* (because it starts with `#`) and is ignored by Algae.

In the second line we define the Fahrenheit temperatures for our table. The expression `0:300:20` gives a vector that goes from 0 to 300 in steps of 20. We add two more elements, 32 and 212, to that vector by using the comma operator. Then we use the `sort` function to sort it in increasing order. Finally, our vector of Fahrenheit temperatures is assigned to the variable `fahr`. Nothing is printed by this statement, since it ends with a semicolon.

The third line computes the Celsius equivalents using the familiar conversion formula. First 32 is subtracted from each element of `fahr`. Then each element is multiplied by 5/9. The result is assigned to the variable `celsius`. Here, again, nothing is printed.

Even though all of the terms so far have been integers, `celsius` is a real-valued vector. Unlike some programming languages, integer division in Algae does not result in truncation—the expression `5/9` returns a real value that is approximately 0.556.

The last line prints the temperatures out as a matrix, with the Fahrenheit temperatures in the first column and their Celsius equivalents in the second column. We do that by using brackets to form a matrix with `fahr` in the first row and `celsius` in the second row and then transposing the matrix with the `'` operator. The question mark on the end is not necessary, since the matrix would also be printed without it, but it does call attention to the line as one that prints.

3.2 Reading and Summing Numbers

This Algae program reads a bunch of numbers, adds them up, and prints the result:

```
# Read and sum some numbers
sum = 0;
$read = 1;
while ($read) { sum += readnum(); }
sum?
```

The program reads numbers one at a time and adds them to `sum`, so in the first line `sum` is initialized to zero. The `readnum` function is used to read the numbers. This function is one of the few standard functions that has a side effect: it assigns to the variable `$read` the number of values that it read. It will be less than the number you asked for if `readnum` runs out of data.

The `while` loop executes its statements repeatedly until its condition tests true. We set `$read` to 1 (true) first, so we know that the `while` loop will execute at least once. After that, the `while` loop continues until `$read` is eventually set to 0 by the `readnum` function. When the `while` loop is done, the last line prints `sum`.

Usually, we give the `readnum` function two arguments: a "shape" vector and a file name. The "shape" vector describes the number of values to read and in what form (vector, matrix, etc.) they should be returned. The values are read from the named file. Since the program above gives neither argument, `readnum` simply reads one value from the standard input device.

The `+=` operator adds the value on the right to the variable on the left; the expression `sum+=readnum()` does the same thing as `sum=sum+readnum()`. Every time the `while` loop repeats, `sum` is increased by the value returned by `readnum`. When `readnum` runs out of data, it returns 0 (so `sum` isn't changed) and sets `$read` to 0 (so the `while` loop quits looping).

Unless you redirect standard input, this program will quietly read what you type until you signal an end-of-file (*C-d* on UNIX systems and *C-z* on VMS).

3.3 Integer Powers

This program illustrates the use of a function. In it, we define the function `power` that computes integer powers of its argument. We also show some Algae code that calls `power`.

```
# power:  raise ‘x’ to the ‘n’-th power
power = function (x; n)
{
  local (y);
  y = x;
```

```
  while (n > 1)
  {
    y = y * x;
    n -= 1;
  }
  return y;
};

power (2; 8)?
power ([ 1, 2; 2, 3 ]; 4)?
```

The first line of this program is an ordinary assignment statement, which assigns the function on its right side to the variable `power`. The function's arguments are defined in the parentheses that follow the `function` keyword. In this case, there are two arguments: `x` and `n`.

In Algae, all function arguments are called "by value". When a function is called, it gets copies of what the calling program gave as arguments and can modify them without changing the values retained by the calling program. In the `power` function we decrement `n` each time through the `while` loop, but this has no effect on the calling program.

The `local` statement defines its argument, the variable `y`, as having local scope. This means that `power` has its own temporary variable called `y` that is defined only within that function. This `y` is first given the value of `x` and then, inside the `while` loop, repeatedly multiplied by `x` until `n` is no longer greater than 1.

The value that `power` computes is returned to its caller by the `return` statement. Any expression may follow `return`, and any number of `return` statements may appear in a function. If the function has no return statement, it returns NULL to its caller.

The first time we call `power`, it returns (and the calling program prints) the value 256. In this case `power` is really not very useful, since we could have obtained the same result with the expression `2^8` using Algae's exponentiation operator.

The `power` function does have a use, though, as the second call to it shows. Algae's exponentiation operator performs in an element-by-element sense, so `[1,2;2,3]^4` returns

```
[          1          16 ]
[         16          81 ]
```

In contrast, `power` performs in a matrix sense. In the program above, it returns

```
[         89         144 ]
[        144         233 ]
```

The algorithm used in the `power` function above can be improved upon by using successive squaring:

```
power = function (x; n)
{
  if (n == 1)
  {
    return x;
  elseif (n%2)
    return x * self (x; n-1);
  else
```

```
      x = self (x; n/2);
      return x * x;
    }
  };
```

This `power` function might be useful, but it's certainly not ready for prime-time. Notice that it gives wrong answers if `n` is 0 or a negative integer. What happens for the expression `power(4;1/2)`, in which `n` is not an integer? To make `power` more robust you'd probably want to add some `if` statements to check for illegal arguments.

4 The Algae Language

In Algae, variables and operators combine to form expressions and statements. The rules for this are mostly conventional; for example, the statement `a=b+c` means that the sum of `b` and `c` is assigned to the variable `a`. In Algae, however, the concept of addition (as well as many other operations) is expanded to include operations between a variety of data classes.

4.1 Entities

The basic unit of data in Algae is the *entity*. Every entity has an attribute called its *class* which describes it as a "scalar", "vector", "matrix", etc. The built-in function `class` returns the class of its argument as a character string. Thus

```
x = 1.2E3; class(x)?
```

prints "scalar" and

```
class (cos)?
```

prints "function". Naturally, the statement

```
class (class (log))?
```

prints "scalar", since `class` returns a scalar character string.

Entities also contain *members*, in which various additional attributes of the entity are stored. For example, matrix entities have members such as `type`, `density`, and `symmetry`. These members are themselves entities, so it is not unusual to have several levels of membership (a member of a member of a member). See Chapter 5 [Data], page 31, for a description of the various classes.

The "dot" operator is used to refer to members. The statements

```
x = 1; x.type?
```

print "integer", the value of the `type` member in `x`. The line

```
foo = 1492; foo.bar?
```

prints "NULL", since scalars don't contain a `bar` member when they're created. You could give it one, of course:

```
foo.bar = "Columbus"
```

4.2 Variables

A *variable* is a symbol that refers to a named entity; the entity is known as the symbol's *value*. A variable comes into being when a value is assigned to it; there are no explicit declarations.

4.2.1 Variable Names

Variables have names consisting of letters, digits, dollar signs ($) and underscores (`_`). A variable name cannot begin with a digit. Variable names are case sensitive, so the variables `foo` and `Foo` are distinct.

We generally use names that begin with $ to refer to global variables that are set at startup or that have some kind of side effect. For example, the variable `$beep` acts like any other variable, but also control's Algae's sound effects. The `who` function does not report variables that begin with *$* unless you give it the argument `"$"`.

One variable name is special—the variable `$$` refers to the global symbol table. For example,

```
a = "x";
$$.(a)
```

prints the value of the global variable `x`. You can use `$$` like any other variable, except that you cannot reassign its value.

The only limitation on the length of a variable name is that it must have at least one character. Short names are convenient for interactive work, but longer, descriptive names have advantages in functions and scripts.

4.2.2 Evaluation of Variables

When a variable name is encountered in an expression, it is evaluated by replacing the variable name with the corresponding named entity. If it is an argument to a function, then a copy is made (conceptually, if not in fact) and passed to the function. Thus, in a function call like `func(a)` (see Section 4.6 [Functions], page 29), the function `func` only gets a copy of `a` and can't modify the `a` belonging to its caller.

There is one exceptional case, involving the "dot" operator, in which an identifier is not evaluated as a variable name. An identifier on the right side of a member expression is taken literally as the name of the member. For example, the statement

```
v.type
```

prints the value of `v`'s member called `type`. You could well have a variable named `type`, but it would not affect the member expression above.

Alternatively, the statement

```
v.(type)
```

has an expression to the right of the "dot" operator, not a simple identifier, and so prints the value of the member of `v` named by the variable `type`.

4.2.3 Scope of Variables

Variables referenced from within functions are global in scope unless they are arguments to the function or are declared local with the `local` statement. For example, the function

```
init = function ()
{
  i = sqrt (-1);
  e = exp (1);
  pi = acos (-1);
};
```

could be used to initialize some commonly used variables. Once you executed `init()`, the variables `i`, `e`, and `pi` would be globally defined.

At the same time, you could have another function

```
egg_hunt = function (v)
{
  local (i);
  for (i in 1:v.ne)
  {
    if (v[i] == 0) { return i; }
  }
}
```

in which `i` has local scope. Here, you can't modify or even get the value of the global variable `i`.

The form of the `local` statement is similar to a function call, and multiple variables may be specified by separating them by semicolons.

The `local` statement is unusual, in that it is a directive to the parser rather than something that is coded. This means that it takes effect when the parser sees it, not when it's executed. In the code

```
f = function ()
{
  a = 1;
  if (0) { local (a); }
  a = 2;
}
```

the global variable `a` gets set to 1. Even though the code in the `if` block is not executed, after seeing it the parser considers `a` to be a local variable. Normally, one simply puts `local` statements at the beginning of a function.

The `veil` statement allows you to make a temporary change to a global variable which will remain active from that point until the end of the current dynamic scope, typically until the function ends. For example, the function

```
prt = function (x)
{
  veil (pi);
  pi = "apple";
  bake ();
};
```

changes the value of the global variable `pi` while it's executing, but `pi` reverts to its previous value when `prt` finishes. If the function `bake` referred to `pi`, it would use or modify the temporary value.

By dynamic scope, we refer to the current execution unit. Within a function, that lasts as long as the function is executing. Otherwise, the dynamic scope is the current file, except that for the `eval` and `exec` functions it is the argument string itself. Also, if executing in interactive mode and not in a function, then the dynamic scope extends only to the end of the last statement on a line.

Unlike the `local` statement, the effect of the `veil` statement takes place at the time it is executed. Once a variable is veiled (and before you set it to something else), it has the same value as the associated global variable.

4.2.4 Predefined Variables

Several variables are already defined for you when Algae starts up. Some of these are used to control Algae, and some provide you information. The predefined variables are:

`$beep` If this variable is "true" (that is, `test($beep)` returns 1), then error messages include a BEL character. This causes most terminals to beep. On startup, `$beep` is 0.

`$digits` This variable specifies the number of significant digits that Algae prints for real and complex numbers. On startup, `$digits` is 4.

`$pid` On startup, `$pid` contains the "process id" of the Algae process.

`$program` On startup, `$program` gives the name of the current Algae program. This will normally be `"algae"`. It would be different if, for example, you had changed the name of the Algae executable.

`$prompt` This variable controls Algae's interactive prompt. It is expected to be a character vector with two elements—on startup it is

```
( "> ", "  " )
```

The first element is Algae's first-level prompt. When Algae is waiting for another line of input before it executes what it has already seen, it presents its second-level prompt. The second element of `$prompt` defines this second-level prompt. If Algae can't make sense of `$prompt`'s value, it does without a prompt.

`$read` The `$read` variable is set as a side-effect of the `readnum` function. It reports the number of values read in the last call to `readnum`.

`$term_width`
This variable tells Algae the width of your terminal in characters. Algae will attempt to adjust its display to fit within this width. On startup, Algae will attempt to sense this on its own. If `term_width` is set to 0, Algae will not wrap lines.

`help` This is simply a character scalar that provides a few basic instructions for using Algae.

You may wish to set some of these variables (like `$beep` or `$prompt`) every time you use Algae. This is conveniently done by putting the assignments in the '`.algae`' file in your home directory.

4.3 Operators

Algae has unary, binary, and ternary operators to perform a variety of operations. The standard arithmetic operators ('`+`', '`-`', '`*`', '`/`', etc.) are available and have the usual precedence. The caret '`^`' is for exponentiation: 8^2 equals 64. The single quote character '`'`' denotes the conjugate transpose.

With a few exceptions, all operators work in the "element by element" sense. For example, if `A` and `B` are matrices with the same size, then `A/B` returns a matrix, each element of which is the quotient of the corresponding elements of `A` and `B`. The '`*`' operator works in an "inner product" sense, so `A*B` is a matrix multiplication. The '`@`' operator is Algae's "element by element" multiplication operator.

When an operator has array operands of different types, one will be automatically converted so that they have a common type, if possible. An `integer` may be converted to `real` or `complex`, and a `real` may be converted to `complex`. No conversion to or from `character` type is possible.

Unlike some other languages, the result of an operation in Algae may have a type that is different than its operands. For example, the result of `1/2` has `real` type, even though its operands have type `integer`. Another example is `(-1)^0.5`, which returns a complex number.

With the exception of `*`, if a binary operator has both a `vector` and a `matrix` as its operands the `vector` is converted to a `matrix` before the operation is performed. Upon conversion, a vector becomes a matrix with one row. For example, in the statement

```
( 1, 2 ) + [ 3, 4 ]
```

the left-hand operand of `+` is the vector `(1,2)`. This is converted to a `matrix` before being added to `[3,4]`.

The following table summarizes the precedence and associativity of Algae's operators. Those shown on the same line have equal precedence, and the rows are in order of decreasing precedence.

operators	**associativity**
`() [] .`	left to right
`'`	right to left
`^`	right to left
`! + -`	(unary) left to right
`* / % @`	left to right
`+ -`	(binary) left to right
`< > <= >= == !=`	left to right
`&`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`:`	left to right
`,`	left to right
`= += -= *= /= @= %=`	right to left

4.3.1 Function Calls

Functions are called in the usual way: the function reference is followed by a list of arguments. For example,

```
w = union (u; v);
```

calls the function `union`, passing it the values of `u` and `v` for arguments. The value returned by the function's `return` statement (or NULL if it doesn't have one) is the value of the function call.

The arguments may be any expressions, and are separated by semicolons. The called function gets only the values of the expressions you give it as arguments, so it can't modify the variables that you give it.

From its definition, the function knows how many arguments to expect. Passing too many arguments is an error. If you pass fewer arguments than the function definition calls for, NULL's are passed in place of the missing ones.

Besides their use in function calls, parentheses are used for several other purposes, including grouping expressions, `if`, `for`, and `while` conditions, function definitions, etc.

4.3.2 Element References

The element reference operation returns specified vector elements or matrix rows and columns. Applied to a table, the operation is performed on each of the table's members.

Elements are referenced by following a vector or matrix expression with a bracket-enclosed list of the desired elements. For vector `v`, the expression `v[w]` gives the elements specified by `w`. The expression within the brackets is expected to be a scalar or a vector—if it's a matrix, it will be converted into a vector if possible. So, for example,

```
v[3:8]
```

prints the third through eighth elements of `v`. If the vector inside the brackets has a numeric type, then it is converted to integer if necessary (by taking the real part and rounding) and then used to refer to the element numbers.

The class of the element reference expression is determined by the class of the specifiers. Once any matrix specifiers are converted to vectors, the dimension of the result is equal to the sum of the dimensions of the specifiers. For example, `M[1;2]` is a scalar, `M[[1];2]` and `M[1;[2]]` are vectors, and `M[[1];[2]]` is a matrix. Notice that the class of the result does not depend on the class of the original entity.

If the vector inside the brackets has character type, then it is taken to refer to the element labels instead of the element numbers. For example, if you set `x` as

```
x = 1:3;
x.eid = "this", "that", "other";
```

then `x["that"]` returns 2, the value of the element of `x` having the label `"that"`. If the labels do not have character type, they are temporarily converted to character type for the comparison.

Element references work the same way for matrices, except that both rows and columns are specified. For example,

```
M[ 1,3; 7:12 ]
```

specifies rows 1 and 3, columns 7 through 12, of the matrix `M`.

The specifiers need not be irredundant. For example, `M[1,1,1;]` returns three copies of the first row of `M`.

Besides their use for element references, brackets are also used to form matrices.

4.3.3 Member References

Members of an entity are referenced with the "dot" operator. For example, `x.type` returns the value of `x`'s member `type`. Notice that, if the right-hand operand is an identifier, then it is taken literally as the name of the desired member. You might have a variable called `type`, but that is irrelevant when it comes to evaluating `x.type`.

On the other hand, if the right-hand operand is an expression, then its value (converted to a character scalar) is taken as the desired member name. Since you can change an identifier into an expression simply by enclosing it in parentheses, the expression `x.(type)` does use the value of the variable `type` as the name of the member.

Member names do not share the same limitations as variable names. In fact, any string of ASCII characters (excluding NUL) will work. This can be pretty useful. For example, you could set up a "vector" of entities (of any class) as in

```
V = {};
V.(1) = A;
V.(2) = B;
V.(3) = C;
V.(4) = D;
```

and then refer to the individual "elements" (that is, the members of `V`) by number. You could handle multiple dimensions by referring to element `"3,2,4"`, for example.

4.3.4 Transpose

The transpose operator applies the conjugate transpose operation to a matrix. For integer, real, and character types, this means moving every element from row `i` and column `j` to row `j` and column `i`. If the matrix has complex type, the complex conjugate operation is applied as well.

If you want the transpose of a complex matrix `M`, and not its conjugate transpose, then use the expression `conj(M')`.

If transpose is applied to a scalar or vector, the entity is first converted to a matrix and then transposed. For example, `(1,2)'` is the same as `[1;2]`—the vector is first converted into a matrix with one row and then transposed to form a matrix with one column.

If this operator is applied to a table, then the operation is performed on each member of that table.

4.3.5 Power

The "power" operator '`^`' is a binary operator that raises its left operand to the power of its right operand. Thus `2^3` gives 8. It associates right-to-left, so the expressions `x^y^z` and `x^(y^z)` are equivalent.

When vectors and matrices are involved, the "power" operator performs in an "element-by-element" sense. For example, if `M` is a matrix, then `M^2` squares each element of `M`. This is definitely not the same thing as `M*M`!

In an expression such as `2^M`, where the left operand is scalar and the right operand is a vector or matrix, the result has each element `i` of `M` replaced by `2^M[i]`. For example,

```
2^(0:3)
```

prints the vector `(1,2,4,8)`.

Notice that the precedence of '`^`' is higher than '`-`', so the expression `-1^2` returns `-1`.

In mathematical usage, `0^0` is undefined—it yields an error in Algae.

If both left and right operands are arrays, then they must have matching dimensions and labels.

If this operator is applied to a table, then the operation is performed on each member of that table.

4.3.6 Not

The '`!`' operator is a unary, prefix operator that returns 1 if its operand is considered "false", and 0 otherwise. The "false" operands are:

- NULL.
- Numeric entities in which every element is zero.
- Character entities in which every element is `""`.
- Vectors and matrices with no elements.
- Tables with no elements.

If this operator is applied to a table, then the operation is performed on each member of that table.

4.3.7 Negation

The unary negation operator '`-`' multiplies its numeric argument by `-1`. The '`+`' operator is for user convenience—Algae ignores it in its unary context. For example, `+-1` gives a negative one. On the other hand, `1+-1` and `1-+1` both return 0.

If one of these operators is applied to a table, then that operation is performed on each member of the table.

4.3.8 Multiplication

Both the '`*`' and the '`@`' operators perform multiplication. The difference is that '`@`' performs in an element-by-element sense, while '`*`' performs in an inner product sense. The operands of '`@`' must have matching dimensions and labels; each element of its left operand is multiplied by the corresponding element of the right operand. For example,

```
( 1, 2, 3 ) @ ( 4, 5, 6 )
```

returns the vector `(4,10,18)`.

When '`*`' is applied to matrices, the number of columns of the left operand must equal the number of rows of the right operand, and the corresponding labels must match. The result is a matrix that has the same number of rows as the left operand and the same number of columns as the right operand.

If one of the operands of '*' is a vector and the other is a matrix, the vector is (conceptually) converted to a matrix before the multiplication is performed. If on the left, it is converted to a matrix with one row; if on the right, it is converted to a matrix with one column.

Multiplication of two vectors gives their inner product.

The '/' operator performs division. Like '@', it performs in an element-by-element sense.

If either operand of any of these operators is a scalar, then the operation is performed in an element-by-element sense. For example

```
2*(1,2,3)
```

gives `(2,4,6)`—every element of the vector is multiplied by the scalar.

The '%' operator performs the modulus operation, producing the remainder when the left operand is divided by the right operand. For example, `(2,2.5,3)%2` returns the vector `(0,0.5,1)`.

The result has the same sign as the left operand. If this operator is applied to a table, then the operation is performed on each member of the table.

4.3.9 Addition

The "addition" operators are the binary operators '+' and '-'. If their operands are numeric, then they perform the normal addition and subtraction operations.

When the '+' operator is applied to character strings, it catenates them. The '-' operator is not defined for character strings.

Between two tables, '+' combines their members in the resulting table. Conceptually, the members of the left operand are inserted in this new table first, followed by the members of the right operand. This means that if the operands have a member with the same name, the value of that member in the resulting table comes from the right operand. For example, if `t` is a table that contains the member "foo", then the result of

```
t + { foo = NULL }
```

is identical to `t` except that its member "foo" has the value NULL.

When the '-' operator is applied to tables, the result is a table that has all the members of the left operand except those that are also in the right operand. For example

```
{x;y;z}-{x;y}
```

returns a table that has the single member "z".

If just one of the operands is a table, then the operation is performed between the other operand and each member of the table.

4.3.10 Relation

The relation operators '<', '>', '<=', '>=', '==', and '!=' return "true" (1) or "false" (0) to reflect the truth of the expression. For example,

```
(1:5) < 3
```

returns `(1,1,0,0,0)`.

If both operands are arrays, then their dimensions and labels must match.

Unlike most of the operators, '`==`' and '`!=`' allow NULL as an operand. In that case, the other operand is simply checked to see if it is or is not a NULL.

If one of these operators is applied to a table, then that operation is performed on each member of the table.

4.3.11 And

The '`&`' operator performs the logical "and" operation in an element-by-element sense. For example,

```
x = 1:5;
x > 2 & x < 4
```

prints `(0,0,1,0,0)`.

If this operator is applied to a table, then the operation is performed on each member of that table.

4.3.12 Or

The '`|`' operator performs the logical "or" operation in an element-by-element sense. For example,

```
x = 1:5;
x < 2 | x > 4
```

prints `(1,0,0,0,1)`.

If this operator is applied to a table, then the operation is performed on each member of that table.

4.3.13 Short And

The '`&&`' operator performs the "short-circuit" logical "and" operation. Both operands are evaluated for "truth" as if they were the test of an `if` statement. If the left operand is "false", then the right operand is not evaluated. If both operands are "true", the result of the operation is 1; otherwise the result is 0. Note that this operation is quite different from that of the '`&`' operator, which works element-by-element.

The following example code is from the `solve` function:

```
if (options != NULL && members (options) == "pos")
{
  A = chol (A);
else
  A = factor (A);
}
```

The variable `options` is first checked to see if it's NULL. Only if it is not NULL is the `member` function called. That's just what we want, since it would be an error to call `member` with a NULL argument.

4.3.14 Short Or

The '`||`' operator performs the "short-circuit" logical "or" operation. Both operands are evaluated for "truth" as if they were the test of an `if` statement. If the left operand is "true", then the right operand is not evaluated. If either operand is "true", the result of the operation is 1; otherwise the result is 0. Note that this operation is quite different from that of the '`|`' operator, which works element-by-element.

In the statement

```
if (options == NULL || options.tol == NULL) { tol = 1e-6; }
```

the variable `options` is first checked to see if it's NULL. Only if it is not NULL is the member reference `options.tol` evaluated.

4.3.15 Generate

Numeric vectors may be generated with the '`:`' operator. An expression like `i:j` generates a vector that starts with the value of `i`. If `j` is greater than `i`, each successive element is 1 greater than the previous one, with the last element less than or equal to `j`. If `j` is less than `i`, each successive element is 1 less than the previous one, with the last element greater than or equal to `j`.

The '`:`' operator also has a ternary form, as in `i:j:k`. This does the same thing as `i:j` except that successive elements differ by `k` instead of 1.

For complex operands, the operation can be described conceptually in the complex plane. A line is drawn between the points `i` and `j`. Then the resulting vector contains the points located on that line segment beginning at `i`, proceeding toward `j`, spaced a distance `k` (or 1, if `k` is not given) apart.

4.3.16 Append

The '`,`' operator "appends" its operands. If the operands are vectors, then the result is a new vector containing both the operands. If the operands are matrices, then the result is a new matrix containing the left operand on the left and the right operand on the right.

If this operator is applied to a table, then the operation is performed on each member of that table.

4.3.17 Assign

The assignment operators are '`=`', '`+=`', '`-=`', '`*=`', '`/=`', '`@=`', and '`%=`'. The '`=`' operator returns the value of the right operand, setting the left operand to that value in the process. For example,

```
a = b = c = 1;
```

assigns 1 to the variables `a`, `b`, and `c`. The assignment operators associate right to left, so first `c` is given the value 1. The result of that expression, `c = 1`, is 1, so that value is used with `b` as if it read `b = 1`.

Notice that a test like

```
if (i = j) ...
```

is entirely different than

```
if (i == j) ...
```

In the first example, `i` is assigned the value of `j` and then that value is tested by the `if` statement. In the second example, the `if` statement tests the equality of `i` and `j`. So if `i` is 1 and `j` is 2, the first example tests true (as well as changing the value of `i`) and the second one tests false.

Individual elements of an array may be changed; for example

```
x[3;4:6] = 0, 0, 0;
```

sets to 0 the elements of `x` in row 3 and columns 4 through 6. The right operand is converted to the same type as the left operand, if possible, and the dimensions must agree. However, if the right operand is a scalar, then it is filled to the appropriate size. Thus, the previous example could just as well be written

```
x[3;4:6] = 0;
```

One more thing about array assignments: If the variable on the left is a vector, but you specify two dimensions for it, then that variable will be converted into a matrix. Likewise, if the variable on the left is a matrix, but you specify only one dimension, then it will be converted into a vector. For example, in the code

```
A = B = [ 1, 2, 3 ];
A[2] = B[;2] = 9;
```

`A` and `B` both begin as matrices. The second element of each array is then changed to 9. In the end, though, `A` is changed to a vector because only one dimension was given for it. `B` remains as a matrix.

The other assignment operators ('`+=`', '`-=`', etc.) are simply for convenience. The expression `i+=j` means the same thing as `i=i+j`, and `i-=j` means the same thing as `i=i-j`. If the left-hand side is an expression, you should keep in mind that it will be evaluated twice. For example,

```
x[ func() ] += 1
```

will actually make two calls to the function `func`.

4.4 Expressions

Assignments are made in the normal fashion: `a=5` sets the variable '`a`' to contain the scalar 5. Variables are not declared prior to use, but rather take on the structure and type of whatever is assigned to them. You could, for example, enter

```
a=1;
a=[a,a];
```

in which case the variable '`a`' is first a scalar and then a matrix. A list of the previously defined variables (except functions) is given by the '`who()`' function, and a variable can be deleted by assigning NULL to it.

In addition to simple variables, elements and members can also be specified on the left hand side of an assignment. For example, the statement

```
A[ 1:4; 1:4 ] = 3;
```

assigns 3 to all of the elements of the first 4-by-4 partition of matrix '`A`', leaving the other elements of `A` unchanged. Values can be assigned to members in the same way. For example, the statements

```
A = [ 1,2,3; 4,5,6 ];
time = [ 1.1, 2.2, 3.3 ];
A.rid = [ "first"; "second" ]
A.cid = time
```

assign the row and column labels of 'A'.

4.5 Statements

Statements are terminated by a question mark, a semicolon, or a newline. Using a question mark or newline causes the value of the statement to be printed; with a semicolon the printing is suppressed. A newline terminates a statement only if it does not occur within parentheses, brackets, or braces.

Statement termination is implied by the closing brace of an "if", "for", or "while" statement or "function" expression. Printing is enabled when the terminator is implied. For example,

```
for (i in 1:10) { i }
```

prints all of the integers from 1 to 10.

When real or complex numbers are printed, Algae prints 4 significant digits by default. This can be changed with the `digits` function, or just by setting the global variable `$digits`.

Comments are introduced with the '#' character. That character and any that follow it on the same line are ignored (excluding the newline).

4.6 Functions

Functions are defined by a function expression, which consists of the keyword "function", followed by a parenthesized list of arguments, followed by a set of statements enclosed by braces. For example, Algae's `max` function is defined by something like

```
max = function (x) { return x[imax(x)]; };
```

Functions are just another class of entity; the statement above defines a function and then assigns it to the variable `max`.

The arguments are variables that are local to the function. On entry, they take on the values of the formal arguments in the calling expression. Passing more arguments than are in the function's definition is an error. If fewer arguments are passed, then NULL's are passed in place of the missing arguments.

The function reference needn't be an identifier—a function expression works just fine. For example,

```
child = (his_functions + her_functions).reproduce();
```

would combine the tables `his_functions` and `her_functions`, reference the member `reproduce`, call it with no arguments, and then assign the result to `child`. Notice that, since '.' and '()' have the same precedence, their left-to-right associativity causes the member reference to occur first and then the function call. In an expression like `sin(1).type`, the function call is performed first.

When a function is compiled, information is included that relates its operations to the file name and line numbers of the source code. If an error occurs, this information is useful in tracking down its cause. In some cases, this information is undesirable and may be removed with the `strip` function. The `message` function is such a case.

5 Data

Entities are the basic objects (constants and variables) in Algae. They contain members and elements. Members contain information about the entity, such as the number of rows in a matrix. An entity's elements, if it has any, contain the data such as the value of a scalar. There are five classes:

`scalar`
: The `scalar` data structure contains a single element, its value, which may have any of the data types described above. Its members are:

  `class`
  : The string `"scalar"`.

  `type`
  : The type (like `"integer"` or `"real"`).

  Notice that a scalar is not the same as a one-element vector or matrix.

`vector`
: The `vector` entity contains a one-dimensional array of elements. Its members are:

  `class`
  : The string `"vector"`.

  `type`
  : The type (like `"integer"` or `"real"`).

  `density`
  : Either `"dense"` or `"sparse"`.

  `ne`
  : The number of elements.

  `nn`
  : The number of non-zero elements.

  `eid`
  : The element labels.

  The element labels of a vector are themselves a vector.

`matrix`
: The `matrix` data structure contains two-dimensional arrays of elements. It contains zero or more rows and columns. The members of a matrix are:

  `class`
  : The string `"matrix"`.

  `type`
  : The type (like `"integer"` or `"real"`).

  `symmetry`
  : One of `"general"`, `"symmetric"`, or `"hermitian"`.

  `density`
  : One of `"dense"`, `"sparse"` or `"sparse_upper"`.

  `nr`
  : The number of rows.

  `nc`
  : The number of columns.

  `nn`
  : The number of nonzero elements. For matrices with "sparse_upper" density, this number does not include the elements on the diagonal, all of which are stored. Sorry.

  `rid`
  : The row labels.

  `cid`
  : The column labels.

  The row and column labels of a matrix are vectors. The row label vector has the same number of elements as its matrix has rows, and likewise for the column label.

`table` The `table` object is like a bucket that can contain any number of other objects.

Several functions return tables. For example, the `eig` function returns a table that contains the matrices `values` and `vectors`.

The entities contained in a table are members of it, so they can be extracted using the `.` operator. Thus `eig(A).values` returns the eigenvalues of `A`.

`function`
Functions are entities just like scalars and matrices, and can be operated on in the same manner. Functions have two members: `class`, which is `"function"`, and `ilk`, which is either `"user"` or `"builtin"`.

The `scalar`, `vector`, and `matrix` classes are known collectively as *arrays*. These are the only classes that contain elements, and all have the member `type` that specifies the data type. There are four types:

`integer` On most machines this is a 32-bit integer, like 42 or -273.

`real` This is a floating point number like 3.1415. It is stored in 64 bits on most machines.

`complex` This is a complex number having both real and imaginary parts.

`character`
This is a character string like `"Hello"`.

Scalar constants are specified in a manner similar to that used in other programming languages. Numeric constants can be given in decimal form, such as `32` or `32.0`, or in scientific notation, such as `3.2E1`. In the latter notation, the letters `e` and `E` may be used interchangeably to prefix the exponent. White space (space, tab, etc.) is significant in that context, of course, so that `1.2e+3` (1200.0) is definitely not the same as `1.2e +3` (4.2).

A `character` constant is a sequence of one or more characters surrounded by matching double-quotes. Within the quotes, the backslash character may be used to introduce "escape sequences" for unusual characters like ⟨newline⟩. For example, the string `"\""` is a string containing a single character (the right double-quote). The following escape sequences may be used:

`\a` bell

`\b` backspace

`\e` escape

`\f` formfeed

`\n` newline

`\r` carriage return

`\t` tab

`\v` vertical tab

`\ooo` octal number

`\xhh` hexadecimal number

The escape sequence `\ooo` consists of a backslash followed by one, two, or three octal digits, which are taken to specify the value of the desired character. For example, `\33` is the ASCII "escape" character (which is also given by `\e`). Likewise, the sequence `\xhh` consists of a backslash followed by '`x`', followed by one or two hexadecimal digits, which specify the value of the desired character. For example, `\x1b` also specifies the ASCII "escape" character.

If the character following the backslash is not one of those specified above, then that character is taken literally. For example, `\"` specifies the double-quote character—not the end of the character constant.

Matrices may be generated by specifying their elements within brackets. Commas separate elements within rows and semicolons separate the rows. Thus `[1,2;3,4]` specifies the matrix

```
[ 1  2 ]
[ 3  4 ]
```

A vector may be generated by using either of the forms '`i:j`' or '`i:j:k`'. It is obtained by starting with the value '`i`' and incrementing by '`k`' (or 1 if the first form is used) while remaining between '`i`' and '`j`', inclusive. (Notice that this is not the same format used by MATLAB.) For example, `[1:8:2]` is the same as `[1,3,5,7]`. All elements within a matrix have the same type ('`integer`', '`real`', etc.); conversion will be performed automatically if possible.

The terms within the brackets may be constants, variables, or expressions. For example, `[1+2,3]` is the same as `[3,3]`. Matrices may be involved, as long as their dimensions are appropriate. For example, if the variable '`A`' is defined to be equal to the square matrix '`[1,2;3,4]`', then a new column could be appended to it with the expression `[A,[5;6]]` to yield

```
[ 1  2  5 ]
[ 3  4  6 ]
```

Matrices may be partitioned by specifying the desired row and column numbers within brackets. For example, if '`A`' is a previously defined matrix then `A[1;1]` specifies a scalar having the value of the element in the first row and the first column of '`A`'. The semicolon within the parentheses separates the row specifiers from the column specifiers.

Members of data structures are referred to by using the member operator '`.`'. For example, the number of rows in a matrix is stored in the member `nr`, so `A.nr` returns the number of rows in the matrix `A`. The member operator associates left to right, so that `A.type.type` returns the string `"character"`.

6 The Standard Functions

Functions are called by giving their name followed by a parenthesized list of their arguments. For example, `sqrt(a)` computes the square root of its argument '`a`'. The parentheses are required, whether they contain any arguments or not. Functions may take more than one number and kind of argument.

6.1 Basic Math

abs (*x*) Function

The `abs` function returns the absolute value of its numeric argument. "Absolute value" is synonymous with "magnitude" for complex types. If *x* is non-scalar, every element is replaced by its absolute value.

`x` must be a numeric scalar, vector, or matrix.

acos (*x*) Function

The `acos` function returns the arc cosine of its numeric argument. If it's argument is complex or has a magnitude greater than one, then the result is complex. Otherwise, the result is real. If *x* is non-scalar, every element is replaced by its arc cosine.

The argument *x* must be a numeric scalar, vector, or matrix.

acosh (*x*) Function

The `acosh` function returns the hyperbolic arc cosine of its numeric argument. If it's argument is complex or is less than 1.0, then the result is complex. Otherwise, the result is real. If *x* is non-scalar, every element is replaced by its hyperbolic arc cosine.

The argument *x* must be a numeric scalar, vector, or matrix. Note that `acosh(0)` is not zero, so if `acosh` is applied to a sparse vector or matrix, then a dense array is the result.

arg (*z*) Function

The `arg` function returns the argument (phase angle) of its numeric argument.

See also `abs`.

asin (*x*) Function

The `asin` function returns the arc sine of its numeric argument. If it's argument is complex or has a magnitude greater than one, then the result is complex. Otherwise, the result is real. If *x* is non-scalar, every element is replaced by its arc sine.

The argument *x* must be a numeric scalar, vector, or matrix.

asinh (*x*) Function

The `asinh` function returns the hyperbolic arc sine of its numeric argument. If *x* is non-scalar, every element is replaced by its hyperbolic arc sine.

The argument *x* must be a numeric scalar, vector, or matrix.

atan (*x*) Function

The `atan` function returns the arc tangent of its numeric argument. If it's argument is complex, then the result is complex. Otherwise, the result is real. If *x* is non-scalar, every element is replaced by its arc tangent.

The argument *x* must be a numeric scalar, vector, or matrix.

atanh (*x*) Function

The `atanh` function returns the hyperbolic arc tangent of its numeric argument. If it's argument is complex or has magnitude greater than 1.0, then the result is complex. Otherwise, the result is real. If *x* is non-scalar, every element is replaced by its hyperbolic arc tangent.

The argument *x* must be a numeric scalar, vector, or matrix.

atan2 (*y; x*) Function

The `atan2` function computes an angle corresponding to *y* and *x*, the lengths of the opposite and adjacent sides. This is similar to `atan(y/x)`, but with these differences:

- The signs of *y* and *x* are used to determine the quadrant in which the angle lies. The angle returned is in the range -pi to pi.
- The argument *x* may be zero, as long as *y* is nonzero. This is different than `atan(y/x)`, because it avoids division by zero.
- Complex type for either argument is not allowed.

The arguments must be scalars, vectors, or matrices, with either integer or real type.

ceil (*x*) Function

The `ceil` function returns the ceiling of its numeric argument. The ceiling of *x* is the smallest integer that is not less than *x*. If *x* is complex, the ceiling is applied to both real and imaginary parts. If *x* is non-scalar, every element is replaced by its ceiling. The type of *x* is not changed by `ceil`.

x must be a numeric scalar, vector, or matrix.

conj (*x*) Function

The `conj` function returns the complex conjugate of its numeric argument. If *x* is non-scalar, every element is replaced by its complex conjugate.

x must be a numeric scalar, vector, or matrix.

cos (*x*) Function

The `cos` function returns the cosine of its numeric argument. If *x* is non-scalar, every element is replaced by its cosine.

x must be a numeric scalar, vector, or matrix.

cosh (*x*) Function

The `cosh` function returns the hyperbolic cosine of its numeric argument. If *x* is non-scalar, every element is replaced by its hyperbolic cosine.

x must be a numeric scalar, vector, or matrix.

erf (*x*) Function

The `erf` function returns the error function of its numeric argument. If *x* is non-scalar, every element is replaced by its error function result.

The error function is defined as `2/sqrt(pi)` times the integral from 0 to *x* of `exp(-t^2) dt`. Among its useful properties, the error function is twice the integral of the Gaussian distribution with 0 mean and variance of 1/2.

Complex arguments are not yet implemented.

erfc (*x*) Function

The `erfc` function returns the complementary error function of its numeric argument. If *x* is non-scalar, every element is replaced by its complementary error function result.

The complementary error function `erfc(x)` is simply `1-erf(c)`, but it is computed in a way that avoids round-off errors when *x* is large.

Complex arguments are not yet implemented.

exp (*x*) Function

The `exp` function returns the exponential of its numeric argument. If *x* is non-scalar, every element is replaced by its exponential.

x must be a numeric scalar, vector, or matrix.

floor (*x*) Function

The `floor` function returns the floor of its numeric argument. The floor of *x* is the largest integer that is not greater than *x*. If *x* is complex, the floor is applied to both real and imaginary parts. If *x* is non-scalar, every element is replaced by its floor. The type of *x* is not changed by `floor`.

x must be a numeric scalar, vector, or matrix.

gcd (*x*) Function

This function computes the greatest common divisor (GCD) of the elements of its vector argument. The input is rounded to integer type. The scalar return value is the GCD. Its member "factors" contains an integer vector such that `gcd(x).factors*x` is equal to `gcd(x)`.

See also `lcm` and `primef`.

imag (*x*) Function

The `imag` function returns the imaginary part of its numeric argument. If x is non-scalar, every element is replaced by its imaginary part. The value returned by `imag` has real type.

See also `real`.

integer (*x*) Function

The `integer` function converts the real part of its numeric argument to integer type, rounding it if necessary.

See also `round`.

lcm (*x*) Function

This function computes the least common multiple (LCM) of the elements of its vector argument. The input is rounded to integer type. The scalar return value is the LCM.

See also `gcd` and `primef`.

log (*x*) Function

The `log` function returns the natural logarithm of its numeric argument. If x is non-scalar, every element is replaced by its logarithm.

x must be a numeric scalar, vector, or matrix.

log10 (*x*) Function

The `log10` function returns the base-10 logarithm of its numeric argument. If x is non-scalar, every element is replaced by its base-10 logarithm.

x must be a numeric scalar, vector, or matrix.

primef (*n*) Function

This function computes the prime factors of its scalar argument, returning them as elements of an integer vector. The argument n is rounded to integer, if necessary. If the argument is less than 2, a zero length vector is returned.

See also `gcd`, `lcm`, and `primes`.

primes (*n*) Function

This function generates a vector of prime numbers that are less than or equal to the scalar argument. The argument *n* is rounded to an integer, if necessary. If the argument is less than 2, a zero length vector is returned.

See also `gcd`, `lcm`, and `primes`.

real (*x*) Function

This function returns the real part of *x*. The value returned by `real` has real type.

See also `imag`.

round (*x*) Function

The `round` function rounds its numeric argument to the nearest whole number. If *x* is complex, both real and imaginary parts are rounded. If *x* is non-scalar, every element is rounded. The type of *x* is not changed by `round`.

If Algae has been compiled to use the system's rint(3m) function, then arguments with a fractional part of exactly 1/2 are rounded to the nearest even whole number. Otherwise, such arguments are rounded towards +infinity.

See also `integer`.

sin (*x*) Function

The `sin` function returns the sine of its numeric argument. If *x* is non-scalar, every element is replaced by its sine.

sinh (*x*) Function

The `sinh` function returns the hyperbolic sine of its numeric argument. If *x* is non-scalar, every element is replaced by its hyperbolic sine.

sqrt (*x*) Function

The `sqrt` function returns the square root of its numeric argument. If *x* is complex or negative, then the result is complex. Otherwise, the result is real. If *x* is non-scalar, every element is replaced by its square root.

tan (*x*) Function

The `tan` function returns the tangent of its numeric argument. If *x* is non-scalar, every element is replaced by its tangent.

tanh (*x*) Function

The `tanh` function returns the hyperbolic tangent of its numeric argument. If *x* is non-scalar, every element is replaced by its hyperbolic tangent.

6.2 Arrays

band (*m*) Function

The `band` function computes the upper and lower bandwidths and profiles of the square matrix *m*. The row bandwidth of a particular row is the number of elements to the left of the diagonal in that row, but not including any consecutive zero elements at the beginning of that row. Likewise, the column bandwidth of a particular column is the number of elements above the diagonal in that column, but not including any consecutive zero elements at the beginning of that column. Notice that, for a diagonal matrix, all row and column bandwidths are zero. The lower bandwidth is the largest of all the row bandwidths, and the upper bandwidth is the largest of all the column bandwidths. The lower profile is the sum of all the row bandwidths, and the upper profile is the sum of all the column bandwidths.

This function returns a real vector with four elements. In order, these elements are as follows: lower bandwidth, lower profile, upper bandwidth, and upper profile.

Profile reduction can be very effective in reducing the time and memory required to factor a matrix. The function `gpskca` provides one approach for doing this.

See also `gpskca`.

bdiag (*x; m; n*) Function

The `bdiag` function mimics the `diag` function, but in a "block" sense. The matrix *x* is taken to be comprised of *m* rows and *n* columns of blocks that are themselves matrices. If either *m* or *n* is 0, then the matrix returned has the blocks of *x* on its diagonal and is elsewhere zero. Otherwise, the blocks of the diagonal of *x* are appended together.

An example of the former case is:

```
> M = magic (4)
        [  9   7   6  12 ]
        [  4  14  15   1 ]
        [ 16   2   3  13 ]
        [  5  11  10   8 ]
> bdiag (M; 0; 2)
        [           9          7          .          . ]
        [           4         14          .          . ]
        [          16          2          .          . ]
        [           5         11          .          . ]
        [           .          .          6         12 ]
        [           .          .         15          1 ]
        [           .          .          3         13 ]
        [           .          .         10          8 ]
```

An example of the latter case is:

```
> bdiag ( M; 2; 2 )
        [  9   7   3  13 ]
        [  4  14  10   8 ]
```

x must be scalar, vector, or matrix. After being converted to a matrix, if necessary, the number of rows of *x* must be evenly divisible by *m* (unless *m* is 0) and the number of columns of *x* must be evenly divisible by *n* (unless *n* is 0).

See also `btrans` and `diag`.

btrans (*x; m; n*) Function

The `btrans` function mimics the transpose operator, but in a "block" sense. The matrix *x* is taken to be comprised of *m* rows and *n* columns of blocks that are themselves matrices. The blocks themselves are not transposed, but are simply moved across the diagonal.

For example:

```
> M = magic (4)
        [  9   7   6  12 ]
        [  4  14  15   1 ]
        [ 16   2   3  13 ]
        [  5  11  10   8 ]
> btrans (M; 2; 2)
        [  9   7  16   2 ]
        [  4  14   5  11 ]
        [  6  12   3  13 ]
        [ 15   1  10   8 ]
```

x must be scalar, vector, or matrix. After being converted to a matrix, if necessary, the number of rows of *x* must be evenly divisible by *m* and the number of columns of *x* must be evenly divisible by *n*.

See also `bdiag`.

circshift (*x; s*) Function

The `circshift` function shifts the elements of the array *x*. The arg *s* is a vector—each of its elements specifies the shift distance for the corresponding dimension of *x*. A positive shift is down or right; a negative shift is up or left. If unspecified, the shift distance is zero (meaning no shift). If *x* is a table, the function is applied to each of its members.

For example:

```
> x = fill (3,3; 1:9)
        [ 1   2   3 ]
        [ 4   5   6 ]
        [ 7   8   9 ]
> circshift (x; 1)
        [ 7   8   9 ]
        [ 1   2   3 ]
        [ 4   5   6 ]
> circshift (x; 0,1)
        [ 3   1   2 ]
        [ 6   4   5 ]
```

```
                [ 9   7   8 ]
    > circshift (x; 1,1)
                [ 9   7   8 ]
                [ 3   1   2 ]
                [ 6   4   5 ]
```

combine (*u; v*) Function

The `combine` function appends the vectors *u* and *v*, removing all but the first of any redundant elements. This is exactly like the `union` function, except that the result is not sorted.

cram (*shape; x*) Function

This function uses the elements of *x* to make an entity described by the vector *shape*. If *shape* is NULL or has no elements, a scalar is returned. If *shape* has one element, then a vector of length `shape[1]` is returned. If *shape* has two elements, then a matrix with `shape[1]` rows and `shape[2]` columns is returned.

The elements of *x* are used, in order, to fill the new entity. For matrices, this proceeds by rows. If the entity returned has more elements than *x*, it is padded with zeros or null strings. For example, `cram( 5; 1,2 )` returns the vector `(1,2,0,0,0)`.

If the return value is an array, it may be either dense or sparse. The code will try to make a reasonable choice, but avoids spending much time in deciding. For example, `cram( 1000; 1 )` returns a sparse vector since it is immediately apparent that most of its elements are zero. On the other hand, `cram( 1000; (1:1000)<2 )` is dense even though it would be more efficient to store it sparse; since its second argument is a dense vector, `cram` would need to check most of its elements to realize that.

The `form` function is identical to this function except that the arrays it returns are always dense.

See also `fill`, `form`, and `zero`.

dense (*x*) Function

The `dense` function converts its argument array to dense storage. See also `sparse`.

diag (*x*) Function

The `diag` function performs two different tasks, depending on the class of *x*. If *x* is a matrix, then its diagonal is returned as a vector. If *x* is a vector, then a matrix is returned which has *x* as its diagonal and is zero elsewhere. (If *x* is a scalar, it is returned intact.)

See also `bdiag`.

dice (*s*) Function

The `dice` function takes a character scalar *s* and returns a character vector, each element of which is a single character from *s*. For example, the expression `dice("Go dawgs!")` yields

```
( "G", "o", " ", "d", "a", "w", "g", "s", "!" )
```

See also `split`.

diff (*v*) Function

The `diff` function takes a numeric vector *v* and returns a vector of differences between its elements. If *v* has `n` elements, then the return vector is

```
v[2]-v[1], v[3]-v[2], ..., v[n]-v[n-1]
```

The return vector has one less element than *v*. If *v* has labels, then the return vector contains all but the last one.

One simple use for `diff` is to compute forward-difference approximations for the slope of a curve. If `c` is a vector containing ordinates of the curve in its elements and abscissae in its labels, then the expression

```
diff (c) / diff (unlabel (c.eid))
```

returns an approximation to its slope. (The call to `unlabel` is probably unnecessary in most cases, but it avoids trouble when the labels of `c` themselves have labels.)

exsparse (*x*) Function

This function disassembles a matrix, returning it in its "coordinate" form, suitable for reassembly by the `mksparse` function. A table is returned, containing the members *shape*, *rows*, *cols*, and *values*. The argument matrix *x* must be numeric. Despite the name of the function, it need not be sparse.

See also `mksparse`.

fill (*shape; x*) Function

This function uses the elements of *x* to make an entity described by the vector *shape*. If *shape* is NULL or has no elements, a scalar is returned. If *shape* has one element, then a vector of length `shape[1]` is returned. If *shape* has two elements, then a matrix with `shape[1]` rows and `shape[2]` columns is returned.

The elements of *x* are used, in order, to fill the new entity. For matrices, this proceeds by rows. If the entity returned has more elements than *x*, then the elements of *x* are reused. For example, `fill( 5; "a","b" )` returns the vector `("a","b","a","b","a")`.

When the return value from `fill` is a vector or matrix, it will always be dense. See the `cram` function for sparse return arrays.

The `form` function differs from this function only in that it pads with zeros rather than reusing elements of *x*.

See also `form`, `cram`, and `zero`.

find (*a; b*) Function

The `find` function locates the elements of *b* that have the values given by *a* and returns a vector containing their element numbers. For example, `find( 2,3; 0,1,2,3,4 )` returns the vector `(3,4)`. One common use is in an expression like `A[find(77;A.rid);]`, which returns the row (or rows) of *a* having the row label 77.

If *a* is a scalar, then `find(a;b)` returns the element numbers of *b*, sorted from smallest to largest, for which the corresponding element of *b* is equal to *a*. If *a* is a vector, then `find(a;b)` returns the same as `find(a[1];b),find(a[2];b),...`.

See also `grep` and `lose`.

first (*v*) Function

The `first` function returns the index of the first "true" element in the vector *v*. If none are found, 0 is returned.

See also `find` and `last`.

form (*shape; x*) Function

This function uses the elements of *x* to make an entity described by the vector *shape*. If *shape* is NULL or has no elements, a scalar is returned. If *shape* has one element, then a vector of length `shape[1]` is returned. If *shape* has two elements, then a matrix with `shape[1]` rows and `shape[2]` columns is returned.

The elements of *x* are used, in order, to fill the new entity. For matrices, this proceeds by rows. If the entity returned has more elements than *x*, it is padded with zeros or null strings. For example, `form( 5; "a","b" )` returns the vector `("a","b","","","")`.

When the return value from `form` is a vector or matrix, it will always be dense. See the `cram` function for sparse return arrays.

The `fill` function differs from this function only in that it reuses the elements of *x* rather than padding with zeros.

See also `fill`, `cram`, and `zero`.

full (*x*) Function

The `full` function converts a matrix stored in "sparse_upper" form to the "sparse" form.

gpskca (*m; flag*) Function

The `gpskca` function attempts to find a symmetric permutation of the rows and columns of matrix *m* to reduce either its bandwidth or its profile. If the *flag* argument is "true" (in the sense of the `test` function, then the Gibbs-Poole-Stockmeyer algorithm is used for bandwidth reduction; otherwise, the Gibbs-King algorithm is used for profile reduction. See the description of the `band` function for a definition of these terms.

The return value is the permutation, given as an integer vector with length equal to the order of *m*. Consider the following example interaction:

```
> a = symmetric (magic(6) > 25);
> band (a)
    ( 4, 13, 4, 13 )
> v = gpskca (a);
> b = a[v;v];
> band (b)
    ( 2, 9, 2, 9 )
```

First, the sparse, symmetric matrix *a* is created. The `band` function shows that it has a bandwidth of 4 and a profile of 13. Next, `gpskca` provides a permutation to reduce the profile. The matrix *b* is set to equal *a* but with this permutation applied. Finally, `band` shows that this permutation reduces both the bandwidth (to 2) and the profile (to 9).

The matrix *m* must be either symmetric or hermitian. This function uses the GPSKCA subroutine written by John Lewis.

See also `band`.

grep (*expr; v*) Function

The `grep` function finds the elements of the vector *v* which match the pattern `expr`. The pattern is an extended regular expression which is given to the UNIX function `egrep` to do the searching.

The character strings in *v* must not contain a newline, or the results will generally be wrong.

Here are some examples (user input is preceded by the '`>`' prompt):

```
> grep ("a"; "ab", "ac", "bc")
    ( 1, 2 )

> grep ("9$"; 1:40)
    ( 9, 19, 29, 39 )

> m = magic (3)
    [ 8  1  6 ]
    [ 3  5  7 ]
    [ 4  9  2 ]
> m.rid = "top", "middle", "bottom";
> m[ grep ("top|bottom"; m.rid); ]
    [ 8  1  6 ]
    [ 4  9  2 ]
```

This is a terribly inefficient implementation (maybe someday Algae will have builtin regular expressions), but maybe you'll find it useful.

See also `find` and `lose`.

hermitian (*x*) Function

This function returns the hermitian part of the square, numeric matrix *x*. A hermitian matrix has complex-conjugate symmetry. It's diagonal must necessarily be real. The returned matrix is complex only if *x* is complex.

If *x* has both row and column labels, they must match.

See also `symmetric`.

ident (*n*) Function

The `ident` function returns an identity matrix with *n* rows and columns.

imax (*v*) Function

If *v* is a vector, this function returns the number of the element with the greatest value. If multiple elements qualify, then the number of the first one is returned.

If *v* is a matrix, then a vector is returned, each element giving the row number of the greatest value in the corresponding column. Again, if more than one element qualifies, the first one is given.

If *v* is a table, then the `imax` function is applied to each of its members.

The argument *v* may not be a scalar, and must not have complex type. Note that `imax(v)` is not necessarily equal to `imin(-v)`.

See also `imin`, `max`, `min`, `isort`, and `sort`.

imin (*v*) Function

If *v* is a vector, this function returns the number of the element with the least value. If multiple elements qualify, then the number of the last one is returned.

If *v* is a matrix, then a vector is returned, each element giving the row number of the least value in the corresponding column. Again, if more than one element qualifies, the last one is given.

If *v* is a table, then the `imin` function is applied to each of its members.

The argument *v* may not be a scalar, and must not have complex type. Note that `imin(v)` is not necessarily equal to `imax(-v)`.

See also `imax`, `max`, `min`, `isort`, and `sort`.

isort (*v*) Function

This function sorts the elements of vector *v* in increasing order, and returns a vector containing the corresponding indices. For example, `sort(20,40,10,30)` returns the vector `(3,1,4,2)`. The expression `v[isort(v)]` actually returns the sorted vector *v*, although the builtin function `sort` does this more efficiently. Complex numbers are sorted primarily by real value and secondarily by imaginary value.

See also `sort`, `max`, `min`, and `set`.

label (*x; a; b*) Function

The `label` function assigns labels to vectors and matrices. If *x* is a vector, *a* is assigned as its element labels. If *x* is a matrix, *a* and *b* are assigned as its row and column labels, respectively. If *x* has any other class, an exception is raised.

See also `unlabel`.

last (*v*) Function

The `last` function returns the index of the last "true" element in the vector *v*. If none are found, 0 is returned.

See also `find` and `first`.

linspace (*a; b; n*) Function

The `linspace` function generates a vector of *n* elements, spaced uniformly between *a* and *b*. All three arguments must be (or be convertible to) scalars. The argument *n* must be (or round to) an integer greater than one.

The vector generation operator '`:`' also generates uniformly spaced vectors. With the operator, the number of elements is determined from the specified spacing; with the `linspace` function, the spacing is determined from the number of elements specified. Also notice that the last element of the vector returned by `linspace` is equal to *b*, which is not necessarily the case with the '`:`' operator.

See also `logspace`.

logspace (*a; b; n*) Function

The `logspace` function generates a vector of *n* elements, spaced logarithmically between *a* and *b*. All three arguments must be (or be convertible to) scalars. Both *a* and *b* must be nonzero, and *n* must be (or round to) an integer greater than one.

Logarithmic spacing means that the logarithm of the result vector is a uniformly spaced vector. This also means that the ratio between any two adjacent elements is constant. Note that even if both *a* and *b* are real, the result will be complex if they have opposite signs, and that a complex result will not necessarily form a straight line in the complex plane.

See also `linspace`.

lose (*a; b*) Function

The `lose` function locates elements of *b* that do not have the values given by *a* and returns a vector containing their element numbers. For example, `lose( 2,3; 0,1,2,3,4 )` returns the vector `(1,2,5)`. One common use is in an expression such as `A[lose(77;A.rid);]`, which returns all of the rows of *a* that don't have the row label 77.

If *a* is a scalar, then `lose(a;b)` returns the element numbers of *b*, sorted from smallest to largest, for which the corresponding element of *b* is not equal to *a*. If *a* is a vector, then `lose(a;b)` returns the intersection of the results of `lose(a[1];b)`, `lose(a[2];b)`, etc.

See also `find`.

magic (*n*) Function

The `magic` function returns a magic square of order *n*. (What system would be complete without it?) The elements of a magic square consist of all the integers 1 through *n*^2, arranged so that the row, column, and the two diagonal sums are equal.

n must be greater than 0. No magic square exists for order 2.

matrix (*x*) Function

This function converts its argument *x* to a matrix if possible. Scalars become one-by-one matrices, vectors become matrices with one row, and matrices are unchanged. If *x* is NULL, `matrix` returns a real matrix with zero rows and zero columns.

See also `scalar` and `vector`.

max (*v*) Function

If *v* is a vector, this function returns its greatest value. If *v* is a matrix, then a vector is returned, each element giving the greatest value of any element in the corresponding column.

If *v* is a table, then the `max` function is applied to each of its members.

The argument *v* may not be a scalar, and must not have complex type.

See also `min`, `imax`, `imin`, `isort`, and `sort`.

merge (*x; y*) Function

This function merges two entities, *x* and *y*, with respect to their labels. The result has labels that consist of the union of the labels of *x* and *y*. Its elements come from the corresponding elements in *x* and *y*; they are summed when the same element occurs in both.

This is demonstrated by the following interactive session:

```
> A = fill (3,3; 1:9)
        [  1   2   3 ]
        [  4   5   6 ]
        [  7   8   9 ]
> A.rid = A.cid = 1:3;
> B = fill (3,3; 10:90:10)
        [ 10  20  30 ]
        [ 40  50  60 ]
        [ 70  80  90 ]
> B.rid = B.cid = 2:4;
> merge (A; B)
        [  1   2   3   0 ]
        [  4  15  26  30 ]
        [  7  48  59  60 ]
        [  0  70  80  90 ]
```

If *x* and *y* are not vectors or matrices, they are simply summed.

min (*v*) Function

If *v* is a vector, this function returns its least value. If *v* is a matrix, then a vector is returned, each element giving the least value of any element in the corresponding column.

If *v* is a table, then the `min` function is applied to each of its members.

The argument *v* may not be a scalar, and must not have complex type.

See also `max`, `imax`, `imin`, `isort`, and `sort`.

mksparse (*t*) Function

This function returns a sparse matrix with dimensions and elements given by the members of table *t*. If the member *shape* exists, it should be a vector with two elements specifying the number of rows and columns of the matrix. If it does not exist, the matrix is sized just large enough to include all of the given elements. The members *rows*, *cols*, and *values* must all exist; these are vectors, all with the same length, corresponding elements of which specify respectively the row number, column number, and value of each specified element of the matrix. If an element is not specified, it is zero. If an element is given more than once, its value is the sum of the given values.

The following interactive session provides an example:

```
> mksparse ({shape=3,4; rows=1,2,3; cols=1,3,2; values=10,11,12})
[          10           .           .           . ]
[           .           .          11           . ]
[           .          12           .           . ]
```

The *values* vector must have numeric type. If t is NULL, a real matrix with 0 rows and 0 columns is returned.

See also `sparse` and `exsparse`.

norm (*x; p*) Function

The `norm` function computes the p-norm of x, where p is 1, 2, "frobenius", or "infinity". (The latter two may be abbreviated as "frob" and "inf".) If p is not specified, the 2-norm is used.

For complex x, the 1 and "infinity" norms deal not with the magnitude of each element, but with the sum of the absolute values of the real and imaginary parts.

pick (*x*) Function

This function returns a vector containing the indices of the non-zero elements of the numeric input vector x. For example, the expression

```
V [ pick (V < 0) ]
```

returns all of the negative elements of `V`.

See also `find`.

product (*x*) Function

The `product` function computes the product of the elements of an array. If x is a scalar, it is returned unchanged. If x is a vector, the product of all of its elements is returned. If x is a matrix, a vector is returned, each element of which is the product of the elements in the corresponding columns of x. If x is a table, the `product` function is applied to each of its members.

rand (*shape*) Function

This function generates pseudo-random numbers with a uniform distribution in the range from 0 to 1. Called with no arguments, it returns a scalar. Otherwise, the vector *shape* specifies a vector or matrix; see the `readnum` function for more information.

The random number generator may be "seeded" with the `srand` function. If you don't call `srand`, the seed is based on the system's clock. Note that `rand` and `randn` share the same seed.

See also `randn` and `srand`.

randn (*shape*) Function

This function generates pseudo-random numbers from a normal distribution with zero mean and unit variance. Called with no arguments, it returns a scalar. Otherwise, the vector *shape* specifies a vector or matrix; see the `readnum` function for more information.

The random number generator may be "seeded" with the `srand` function. If you don't call `srand`, the seed is based on the system's clock. Note that `rand` and `randn` share the same seed.

See also `rand` and `srand`.

reverse (*v*) Function

The `reverse` function simply reverses the order of the elements in vector *v*. For example, `reverse(1,2,3)` returns the vector `(3,2,1)`.

scalar (*x*) Function

This function converts its argument *x* to a scalar if possible. See also `matrix` and `vector`.

select (*x*) Function

The `select` function selects one element at random from the given vector. The result is a scalar. The vector *x* must have at least one element.

seq (*n*) Function

The `seq` function returns a vector of consecutive integers from 1 to *n*. The argument *n* is first rounded to the nearest integer; if it is less than 1, a zero-length vector is returned.

shape (*x*) Function

This function returns the "shape" (that is, the dimensions) of its argument *x*. This return value has the same form as is used for the *shape* arguments of functions like *readnum* and *rand*. If *x* is a scalar, NULL is returned. If *x* is a vector, a scalar is returned having the length of *x* as its value. If *x* is a matrix, a vector is returned having the number of rows and the number of columns as its first and second elements, respectively. If *x* has any other class, an exception is raised.

sign (*v*) Function

The `sign` function returns an array with the same size as *v*. Each element of the returned array is either -1, 0, or +1, depending on whether the corresponding element of *v* is negative, zero, or positive, respectively. The argument *v* must be a numeric scalar or array.

sort (*v*) Function

This function sorts the elements of vector *v* in increasing order. For example, `sort(20,40,10,30)` returns the vector `(10,20,30,40)`. Complex numbers are sorted primarily by real value and secondarily by imaginary value.

See also `isort`, `max`, `min`, and `set`.

sparse (*x*) Function

This function converts its argument *x* to sparse storage.

srand (*s*) Function

The `srand` function is used to "seed" the random number generator `rand`. (OK, they're really pseudo-random numbers.) The seed *s* determines the sequence of numbers returned by `rand`. If *s* is NULL (or if `srand` is never called at all), then the seed is taken from the system's clock. See also `rand`.

sum (*x*) Function

The `sum` function sums elements of arrays. If *x* is a scalar, it is returned unchanged. If *x* is a vector, the sum of all of its elements is returned. If *x* is a matrix, a vector is returned, each element of which is the sum of the elements in the corresponding columns of *x*.

surprise (*rows; cols; type; density; symmetry; other*) Function

The `surprise` function returns a more-or-less random matrix with characteristics given by the input arguments. It may be useful for testing or timing purposes.

Each of the arguments to `surprise` may be a vector, in which case one of its members is chosen at random for that characteristic. For example, if *rows* is the vector `4,5,6` then the resulting array will have either 4, 5, or 6 rows.

The *rows* and *cols* arguments specify the number of rows and columns in the array. The choice of dimensions has a lower priority than the other characteristics, so any dimensions that are incompatible with other choices are discarded. For example, consider the call

```
surprise (3,4; 4,5; "real"; ; "general","symmetric")
```

This specifies a real matrix that is either general or symmetric. If general symmetry is chosen, then the array will have either 3 or 4 rows and either 4 or 5 columns. However, if a symmetric array is chosen, then it must be square and the result will have 4 rows and 4 columns.

On the other hand, the specified dimensions can affect the choice of the other characteristics. For example, in the call

```
surprise (3; 4; "real"; ; "general","symmetric")
```

the result will not be symmetric, because the dimensions preclude it.

The *type* argument may contain any or all of the character strings `"integer"`, `"real"`, or `"complex"`. Only numeric types are supported. The default is `"real"`.

The *density* argument specifies the approximate ratio of nonzero to total array elements. For diagonal arrays, only the diagonal elements are considered. The default is 1.0.

The *symmetry* argument may be `"general"`, `"symmetric"`, or `"hermitian"`. The default is `"general"`.

The *other* argument may be `"none"`, `"diagonal"`, or `"positive_definite"`. The default is `"none"`.

symmetric (*x*) Function

This function returns the symmetric part of the square, numeric matrix *x*. If *x* has both row and column labels, they must match.

See also `hermitian`.

tril (*a; k*) Function

The `tril` function returns the lower "triangular" part of the matrix *a*. The return matrix is identical to *a* except that, for any `i`, all elements `a[i;i+k+1]`, `a[i;i+k+2]`, etc. are zero. If *k* is NULL, it defaults to zero. Here is an example:

```
> tril (magic (4); 1)
        [  9   7   0   0 ]
        [  4  14  15   0 ]
        [ 16   2   3  13 ]
        [  5  11  10   8 ]
```

triu (*a; k*) Function

The `triu` function returns the upper "triangular" part of the matrix *a*. The return matrix is identical to *a* except that, for any `i`, all elements `a[i;i+k-1]`, `a[i;i+k-2]`, etc. are zero. If *k* is NULL, it defaults to zero. Here is an example:

```
> triu (magic (4); 1)
        [  0   7   6  12 ]
        [  0   0  15   1 ]
        [  0   0   0  13 ]
        [  0   0   0   0 ]
```

unlabel (*x*) Function

The `unlabel` function removes the labels from its argument. If *x* is a vector or a matrix, it sets the labels to NULL and returns the result. If *x* has any other class, it is returned unchanged.

vector (*x*) Function

This function converts its argument *x* to a vector if possible. Scalars become one-element vectors. If *x* is a matrix with either one row or one column, then it is converted to a vector.

See also `scalar` and `matrix`.

zero (*shape*) Function

This function generates an integer scalar, vector, or matrix, all the elements of which are zero. The vector *shape* specifies the dimensions of the entity returned; see the `readnum` function for more information.

If a vector or matrix is returned, it is stored in sparse form. This may or may not be the most efficient storage for your application; use the `dense` function to convert it if desired.

See also `dense`.

6.3 Sets

complement (*a; b*) Function

The `complement` function returns the relative complement of *a* in *b*; that is, the set of elements of *b* which are not in *a*. The return value is a set, which we define as a sorted, irredundant vector. "Sorted" means sorted by increasing value; character strings are sorted by ASCII values and complex numbers are sorted with the real values as the primary key and the imaginary values as the secondary key.

If *b* has labels, then the returned vector also has labels that correspond to the retained elements. If *b* has no labels, the labels of the returned vector give the index of the corresponding element in *b*.

If *b* has redundant elements, the element retained is not necessarily the first. For example, `complement(1;2,2,2).eid` might be 1, 2, or 3.

See also `intersection`, `set`, and `union`.

intersection (*a; b*) Function

The `intersection` function returns the intersection of set *a* in *b*; that is, the set of elements that are in both *a* and *b*. The return value is a set, which we define as a sorted, irredundant vector. "Sorted" means sorted by increasing value; character strings are sorted by ASCII values and complex numbers are sorted with the real values as the primary key and the imaginary values as the secondary key.

The return vector has no labels.

See also `complement`, `set`, and `union`.

set (*x*) Function

This function changes *x* into a set, if possible. A "set" is defined as a sorted, irredundant vector. "Sorted" means sorted by increasing value; character strings are sorted by ASCII values and complex numbers are sorted with the real values as the primary key and the imaginary values as the secondary key. "Irredundant" means that no two elements have the same value.

If *x* has labels, then the returned vector also has labels that correspond to the retained elements. If *x* has no labels, the labels of the returned vector give the index of the corresponding element in *x*.

If *x* has redundant elements, the element retained is not necessarily the first. For example, `set(1,1,1).eid` might be 1, 2, or 3.

See also `complement`, `intersection`, and `union`.

union (*a; b*) Function

The `union` function returns the union of sets *a* and *b*. The return value is a set, which we define as a sorted, irredundant vector. "Sorted" means sorted by increasing value; character strings are sorted by ASCII values and complex numbers are sorted with the real values as the primary key and the imaginary values as the secondary key.

The return vector will always have labels. To understand them, you need to know that the function call `union(a;b)` is identical to the code `set(a,b)` provided that *a* and *b* are both vectors. (The `union` function will convert them both to vectors if necessary.)

See also `complement`, `intersection`, and `set`.

6.4 Linear Algebra

backsub (*afac; b*) Function

The `backsub` function solves the set of linear equations `A*x=b` for *x*. (Despite the name, it does both forward and backward substitution.) The argument *afac* must contain the factorization of `A` returned as a table from either of the functions `factor` or `chol`. The argument *b* provides the right-hand side.

This function handles both LAPACK (dense) and SuperLU (sparse) factors. It determines the appropriate solution method based on the members in the *afac* table.

If BCSLIB-EXT (a sparse math library produced by Boeing Computer Services) is available on your system and Algae has been compiled for it, then the `backsub` function can use it. If *afac* contains factors computed by BCSLIB-EXT routines (in `factor` or `chol`), then `backsub` will call the corresponding routines to perform the back substitution.

See also `chol`, `factor`, and `solve`.

chol (*m*) Function

The `chol` function computes the Cholesky factorization of the symmetric, positive definite matrix *m*. The factors are returned in a table, which may contain a variety of members depending on the type and density of *m*. For example, if *m* is a real, dense matrix, then `"L"` (the output of the LAPACK routine `DSYTRF` routine) and `"RCOND"` are returned. For now, you'll have to go digging in the code if you really want to understand the output. The idea, though, is that functions like `backsub` can make sense of them so that you don't have to.

Although its other members vary, the table returned by `chol` always contains a member named `"RCOND"`. This non-negative real scalar is an estimate of the reciprocal of the condition number of *m*. If `"RCOND"` is very small (that is, the condition number is very large), then the matrix *m* is ill-conditioned. A warning message is produced if `"RCOND"` is less than 1.0E-8.

The key difference between the `chol` and `factor` functions is that `chol` requires that the matrix be symmetric and positive definite. In that case, no pivoting is required; this generally results in a savings in execution time over the `factor` function.

Unless the matrix *m* happens to be diagonal (or BCSLIB-EXT is used), `chol` uses a dense method from LAPACK. Consequently, `factor` is preferred over `chol` for large, sparse systems.

If BCSLIB-EXT (a sparse math library produced by Boeing Computer Services) is available on your system and Algae has been compiled for it, then `chol` may use its routines to factor sparse matrices. In turn, the `backsub` and `solve` functions can use these factors to solve linear equations. It is often useful to compute the factors of a matrix and then save them (using `put`) for later use. Keep in mind, however, that if these factors were computed with the BCSLIB-EXT routines, then the Algae version with which you intend to use them must also support BCSLIB-EXT. The table returned by `chol` contains the member `HOLD` iff a BCSLIB-EXT routine was used to produce it. BCSLIB-EXT is never used on dense matrices.

See also `backsub`, `factor`, and `solve`.

eig (*a; b; opts*) Function

The `eig` function computes eigenvalues and eigenvectors for the standard and generalized eigenvalue problems

```
A * x = lambda * x
```

and

```
A * x = lambda * B * x
```

If *a* is the only matrix argument, then both *lambda* and *x* are computed for the standard problem. If a matrix *b* is given as the second argument, then the generalized problem is solved.

A table of options *opts* may be given as the last argument. If it contains the member `no_right`, then the right eigenvectors *x* are not computed. If it contains the member `left`, then the corresponding left eigenvectors are computed.

This function returns a table containing the eigenvalues in a member called `values` and, if appropriate, the (right) eigenvectors in a member called `vectors`. If the `left` option was given and the problem is unsymmetric, then the left eigenvectors are returned in a member called `left_vectors`.

The solution method used depends on the type and symmetry of the coefficient arrays. Some of these methods entail strict requirements, such as positive definiteness, on the matrices. Each case is described below, along with the LAPACK routine called to solve it. See the *LAPACK Users' Guide* for more information.

For real, symmetric, standard problems, DSYEV is used. The eigenvalues are real and in ascending order; the eigenvectors are orthonormal.

For complex, hermitian, standard problems, ZHEEV is used. The eigenvalues are real and in ascending order; the eigenvectors are orthonormal.

For nonsymmetric, standard problems, DGEEV is used for real type and ZGEEV for complex type. The eigenvalues are complex, and complex conjugate pairs appear consecutively with the eigenvalue having the positive imaginary part first. The eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

For real, symmetric, generalized problems, DSYGV is used. This method requires that *a* be symmetric and that *b* be symmetric and positive definite. The eigenvalues are real and in ascending order. The eigenvectors are *b*-orthonormal.

For real, nonsymmetric, generalized problems, DGGEV is used. This method performs full balancing on *a* and *b*. The eigenvalues are complex, and complex conjugate pairs appear consecutively with the eigenvalue having the positive imaginary part first. Note, however, that the eigenvalues are returned in a table; the member `num` contains their numerators and member `denom` contains their denominators. The quotients may easily overflow or underflow, and the denominator (and maybe the numerator, too) may be zero. A good beginning reference is the book *Matrix Computations* by G. Golub and C. Van Loan. Each eigenvector is scaled so that the sum of the absolute values of the real and imaginary parts of the largest component is 1, except that for eigenvalues with numerator and denominator both zero, the corresponding eigenvector is zero.

For complex, hermitian, generalized problems, ZHEGV is used. This method requires that a be hermitian and that b be hermitian and positive definite. The eigenvalues are real and in ascending order. The eigenvectors are b-orthonormal.

For complex, nonsymmetric, generalized problems, ZGGEV is used. The eigenvalues and eigenvectors are complex. Note, however, that the eigenvalues are returned in a table; the member `num` contains their numerators and member `denom` contains their denominators. The quotients may easily overflow or underflow, and the denominator (and maybe the numerator, too) may be zero. A good beginning reference is the book *Matrix Computations* by G. Golub and C. Van Loan. Each eigenvector is scaled so that the sum of the absolute values of the real and imaginary parts of the largest component is 1, except that for eigenvalues with numerator and denominator both zero, the corresponding eigenvector is zero.

See also `iram`.

equilibrate (*A*) Function

The `equilibrate` function computes row and column scalings intended to equilibrate a matrix and reduce its condition number. The input matrix A need not be square, but it must be numeric and both dimensions must be nonzero. If A is hermitian or real symmetric, it must be positive definite.

If `equilibrate` succeeds, it returns a table containing the following members:

r A vector containing the row scale factors.

c A vector containing the column scale factors.

rowcnd The (scalar) ratio of the smallest to the largest of the row scale factors.

colcnd The (scalar) ratio of the smallest to the largest of the column scale factors.

amax The (scalar) absolute value of the largest matrix element.

Even if the matrix A is symmetric or hermitian, the row and column scale factors may or may not be identical. If not, the resulting equilibrated matrix would no longer be symmetric or hermitian. You will have to decide whether the improvement in matrix condition is worth the loss of symmetry. The row and column scale factors will be identical for dense, real, symmetric matrices and for dense, complex, hermitian matrices (both of which must also be positive definite); otherwise, there's no guarantee.

Oftentimes, going on to apply the equilibration scale factors is not worth the effort. Roughly speaking, if *amax* is not close to overflow or underflow and both *rowcnd* and *colcnd* are greater than 0.1, it probably isn't worth doing.

If `equilibrate` is unsuccessful, it returns a scalar integer instead of a table. If this value is positive, it indicates the row causing the problem. If it is negative, its absolute value indicates the column causing the problem.

To apply the equilibration, the matrix must be pre- and post-multiplied by the scale factors. If r and c are identical, you'll probably want to use the `transform` function to preserve symmetry. Here is an example usage:

```
e = equilibrate (A);
if (equal (e.r; e.c))
{
  Ae = transform (A; diag (e.r));
else
  Ae = diag (e.r) * A * diag (e.c);
}
```

Notice, though, that this example does not check the result of `equilibrate` to determine if it succeeded, it applies the scale factors without checking to see if it's worth it, and it assumes that A obeys the requirement regarding positive definiteness.

factor (*m*) Function

The `factor` function computes a triangular factorization of the matrix m. The factors are returned in a table, which may contain a variety of members depending on the type, density, and definition of m. For example, if m is a real, dense, symmetric matrix, then `"LD"` and `"IPIV"` (the output of the LAPACK routine `DSYTRF` routine) are returned. For now, you'll have to go digging in the code if you really want to understand the output. The idea, though, is that functions like `backsub` can make sense of them so that you don't have to.

For sparse systems, the SuperLU package is used. These routines reorder the columns of the matrix to increase its sparsity, and perform dynamic pivoting of the rows for numerical stability. (No advantage is taken of symmetry.) For the column ordering, the Multiple Minimum Degree (MMD) algorithm is used if the matrix is symmetric; otherwise, the Column Approximate Minimum Degree (COLAMD) algorithm is used. Currently, there is no provision to supply your own ordering.

Although its other members vary, the table returned by `factor` always contains a member named `"RCOND"`. This non-negative real scalar is an estimate of the reciprocal of the condition number of m. If `"RCOND"` is very small (that is, the condition number is very large), then the matrix m is ill-conditioned. A warning message is produced if `"RCOND"` is less than 1.0E-8.

If BCSLIB-EXT (a sparse math library produced by Boeing Computer Services) is available on your system and Algae has been compiled for it, then `factor` may use its routines to factor sparse matrices. In turn, the `backsub` and `solve` functions can use these factors to solve linear equations. It is often useful to compute the factors of a matrix and then save them (using `put`) for later use. Keep in mind, however, that if these factors were computed with the BCSLIB-EXT routines, then the Algae version with which you intend to use them must also support BCSLIB-EXT. The table returned by `factor` contains the member `HOLD` iff a BCSLIB-EXT routine was used to produce it. BCSLIB-EXT is never used on dense matrices.

The key difference between the `chol` and `factor` functions is that `chol` requires that the matrix be symmetric and positive definite. In that case, no pivoting is required; this generally results in a savings in execution time over the `factor` function.

See also `backsub`, `chol`, and `solve`.

fft (*x; opts*) Function

The `fft` function computes the complex discrete Fourier transform of the vector (or matrix—see below) *x* using the fast Fourier transform method. The result is ordered such that the corresponding frequencies are increasing (from negative to positive). Note that this is a different order than that returned by many FFT routines.

If *x* has `N` elements, the transform is a complex vector with the same length. If `N` is even, then element `k` of the transform is equal to

```
sum (x @ exp (-2*i*pi*(k-N/2)*((1:N)-1)/N))
```

If `N` is odd, then element `k` is equal to

```
sum (x @ exp (-2*i*pi*(k-(N+1)/2)*((1:N)-1)/N))
```

The method is most efficient when the number of elements in *x* is the product of small primes. For example, a vector with 1021 elements (a large prime number) takes roughly 100 times longer to transform than a vector with 1024 elements (a power of 2). On the other hand, 1152 elements (with prime factors of 2 and 3) is almost as fast as 1024.

If *x* has integer or real labels, they are taken to be the corresponding abscissae (i.e., time or distance) and are transformed accordingly to form labels for the results vector. In that case, the label values must be evenly spaced.

If *x* is a matrix, the transform is computed for each row or column. By default this is done by column, but you can change it to work by rows by supplying the `row` option, as described below.

The argument *opts* is an options table; it may contain any or all of the following members. (The values of these members are irrelevant; only their existence is signficant.)

`row` If *x* is a matrix, this option causes the transform of each row to be computed. By default, the columns are transformed.

`estimate` When planning the transform, use a simple heuristic to pick a (probably sub-optimal) plan quickly. This is the default rigor. This and the following options have an effect only if Algae has been linked with the FFTW package.

`measure` Find an optimized plan by actually computing several FFTs and measuring their execution time.

`patient` Like `measure`, but considers a wider range of algorithms. This may produce a “more optimal” plan, but at the expense of a longer planning time.

`exhaustive`
Like `patient`, but with an even wider range of algorithms and substantially increased planning time.

`forget` Plans are kept for use on subsequent FFTs with the same size and characteristics. The `forget` option causes this accumulated information to be discarded.

See also `ifft`.

filter (*b; a; x; z*) Function

The `filter` function computes a digital filter from the standard difference equation

```
y[n] = b[1]*x[n] + b[2]*x[n-1] + b[3]*x[n-2] + ...
                 - a[2]*y[n-1] - a[3]*y[n-2] - ...
```

This filter is implemented using the "direct form II transposed" method. For more information, see Chapter 6 of the book "Discrete-Time Signal Processing" by Oppenheim and Schafer.

ifft (*x*) Function

The `ifft` function computes the complex inverse discrete Fourier transform of the vector (or matrix—see below) *x* using the fast Fourier transform method. The input is ordered such that the corresponding frequencies are increasing (from negative to positive). Note that this is a different order than that required by many inverse FFT routines.

If *x* has `N` elements, the transform is a complex vector with the same length. If `N` is even, then element `n` of the transform is equal to

```
(1/N) * sum (x @ exp (2*i*pi*((1:N)-N/2)*(n-1)/N))
```

If `N` is odd, then element `n` is equal to

```
(1/N) * sum (x @ exp (2*i*pi*((1:N)-(N+1)/2)*(n-1)/N))
```

The method is most efficient when the number of elements in *x* is the product of small primes. For example, a vector with 1021 elements (a prime number) takes roughly 100 times longer to transform than a vector with 1024 elements (a power of 2). On the other hand, 1152 elements (with prime factors of 2 and 3) is almost as fast as 1024.

If *x* has integer or real labels, they are taken to be the corresponding abscissae (i.e., frequencies) and are transformed accordingly to form labels for the results vector. In that case, the label values must be evenly spaced.

If *x* is a matrix, the transform is computed for each row or column. By default this is done by column, but you can change it to work by rows by supplying the `row` option.

For a description of the valid options and their effects, see the `fft` function.

See also `fft`.

inv (*a; opts*) Function

The `inv` function computes the inverse of a matrix. This is sometimes useful, but you should be aware that an alternative approach using `factor` (or `chol`) and `solve` is usually more efficient and accurate. (See below for more details.)

The `inv` function is implemented simply by calling the `solve` function with an identity matrix as the second argument. The argument *opts* is optional and is passed as the third argument to `solve`. For example,

```
inv (A; {pos});
```

computes the inverse of the positive-definite matrix `A`.

The matrix inverse is often used to solve systems of linear equations. If `Ax=b` is such a system, where `A` is a square matrix with full rank, then its solution may be computed as

```
x = inv (A) * b;
```

A more efficient and accurate approach, though, is to use the `solve` function as

```
x = solve (A; b);
```

The use of `solve` is appropriate even when several right-hand sides are to be considered separately. For example, the code

```
x = {};
A_inv = inv (A);
for (b in members (blist))
{
  x.(b) = A_inv * blist.(b);
}
```

would be better written as

```
x = {};
A_fac = factor (A);
for (b in members (blist))
{
  x.(b) = solve (A_fac; blist.(b));
}
```

See also `backsub`, `chol`, `factor`, and `solve`.

iram (*n; op_func; b_func; params; options*) Function

This function is an interface to the ARPACK routines for solving algebraic eigenvalue problems. It is based on the implicitly restarted Arnoldi method. For hermitian problems, this is a Lanczos method. This code is particularly well suited to large, sparse problems.

The `iram` function provides almost all of the capability of ARPACK. However, it is a relatively low-level function and requires very careful use. Because of it's design, many simple input errors cannot be detected by the function and may well result in incorrect results. I hope to soon see a high-level function written, perhaps called `arnoldi`, that calls `iram` but trades off some of its flexibility in exchange for a simplified and less error-prone interface.

To use `iram` effectively, you will need more information regarding the various solution modes than can be included here. Please consult the ARPACK users manual.

The most general problem solved by `iram` is to compute a few eigenvalues (`lambda`) and eigenvectors (`x`) for

```
A * x = M * x * lambda
```

where A and B are real or complex square matrices and M is hermitian (symmetric when real). This is a standard eigenvalue problem if M is the identity matrix; otherwise it is a generalized problem.

The argument *n* specifies the dimension of the problem. This should be an integer scalar, or at least able to be cast to one.

Rather than providing A and B directly, `iram` requires input functions that perform certain operations on those matrices. This allows numerous solution approaches (called "modes"), which are described in detail below. The two user-supplied functions are called as

```
op_func (v; params)
b_func (v; params)
```

The operations required of the *op_func* function differ depending on the solution mode. In the simplest case (mode 1), it is just the matrix-vector multiplication `A*v`. The argument *params* is passed unchanged from the *params* argument to `iram`—it is commonly a table containing the coefficient arrays A and B.

The requirements for the *b_func* function also depend on the solution mode. It is not used at all for some modes, in which case it may be NULL. As with *op_func*, the *params* argument is passed unchanged from the `iram` input.

The *options* argument is a table; its members specify various problem and solution options:

`mode` The integer solution mode, which defaults to 1.

- `1` This is the "regular" mode, for standard problems only. Function *op_func* should return `A*v`, and *b_func* should be NULL. This mode is mostly suited to finding a few of the eigenvalues with largest magnitude. It may be used with any problem type.
- `2` This is the "regular inverse" mode for generalized problems. Function *op_func* should return `inv(B)*A*v` (conceptually, but you should use `factor` and `solve` rather than `inv`), and *b_func* should return `B*v`. This mode may not be used with real, symmetric problems.
- `3` This is the "shift-invert" mode for standard or generalized problems. Function *op_func* should return `inv(A-sigma*B)*B*v` (conceptually, but you should use `factor` and `solve` rather than `inv`), and *b_func* should return `B*v`. This mode may be used with any problem type. Convergence is enhanced to eigenvalues that are near the value of *sigma*. For this mode, the `which` option refers to the inverted problem, so use "LM" to refer to the eigenvalues closest to *sigma*.
- `4` This is the "buckling" mode for generalized problems. Function *op_func* should return `inv(A-sigma*B)*A*v` (conceptually, but you should use `factor` and `solve` rather than

inv), and *b_func* should return A*v. This mode may only be used for real, symmetric problems. Convergence is enhanced to eigenvalues that are near the value of *sigma*. For this mode, the which option refers to the inverted problem, so use "LM" to refer to the eigenvalues closest to *sigma*.

5 — This is the "Cayley" mode for generalized problems. Function *op_func* should return inv(A-sigma*B)*(A+sigma*B)*v (conceptually, but you should use factor and solve rather than inv), and *b_func* should return B*v. This mode may only be used for real, symmetric problems. Convergence is enhanced to eigenvalues that are near the value of *sigma*. For this mode, the which option refers to the inverted problem, so use "LM" to refer to the eigenvalues closest to *sigma*.

bmat
: A character scalar: "I" for standard problems or "G" for generalized problems. The default is "I".

which
: A character scalar that specifies which of the eigenvalues to compute. For other than "regular" mode (mode 1), this is with respect to the transformed problem. For example, with the "shift-invert" mode (mode 3), one usually wants the eigenvectors nearest the shift point. In the transformed problem, they are the eigenvalues with largest magnitude ("LM"). For real, symmetric problems, *which* must be one of the following: "LA" (largest amplitude), "SA" (smallest amplitude), "LM" (largest magnitude), "SM" (smallest magnitude), or "BE" (both ends of the spectrum). For non-symmetric or complex problems, *which* must be one of these: "LM" (largest magnitude), "SM" (smallest magnitude), or "LR" (largest real), "SA" (smallest real), "LI" (largest imaginary), or "SI" (smallest imaginary). The default is "LA" for symmetric problems and "LR" for the others.

nev
: The number of eigenvalues to be computed. This must be greater than zero and less than *n*. The default is the lesser of 10 and n/2. For non-symmetric and complex problems, *nev* must be less than n-1.

ncv
: Controls the number of Arnoldi vectors generated at each iteration. For real, symmetric problems, this must be less than or equal to *n*, must exceed *nev*, and defaults to the lesser of 2*nev and n. For real, non-symmetric problems, *ncv* must be less than or equal to *n*, must exceed nev+1, and defaults to the lesser of 2*nev+1 and n. For complex problems, *ncv* must be less than or equal to *n*, must exceed *nev*, and defaults to the lesser of 2*nev and n.

mxiter
: The maximum number of Arnoldi update iterations allowed. The default is 100.

basis
: If "true" (in the sense of the test function), then iram returns, along with its other results, an orthonormal basis for the associated approximate invariant subspace.

`vectors` If "true" (in the sense of the `test` function), then `iram` returns the eigenvectors along with its other results.

`tol` The stopping criterion. If *tol* is NULL or less than or equal to zero, then machine precision is used.

`resid` If present, this is an n-length vector that provides the initial residual vector. Otherwise, a random initial vector is used.

`shift` A real scalar providing the shift point (*sigma*) for a spectral transformation. This is used in modes 3, 4, and 5. The default is 0.0.

leastsq (*A; b*) Function

This function solves full-rank overdetermined and underdetermined linear systems using a QR or LQ factorization. The inputs are the coefficient array *A* and the RHS *b*. The return value is the solution array *x*.

An underdetermined system is one for which *A* has fewer rows than columns. The rank of *A* must be equal to the number of rows of *A*. The return value is the minimum norm solution.

An overdetermined system is one for which *A* has more rows than columns. This function treats a square *A* as if it were an overdetermined system. The rank of *A* must be equal to the number of columns of *A*. The return value is the solution to the least squares problem

```
minimize || B - A*X ||
```

See also `solve`.

mult (. . .) Function

This function multiplies together a series of matrices, performing the multiplications in an order so as to minimize the total number of operations.

Only the leftmost non-NULL arguments are used, and at most 10 arguments are allowed. They will be converted to matrices if necessary (and possible), but in a context-free manner. A scalar becomes a 1x1 matrix, and a vector becomes a matrix with 1 row. This is in contrast to Algae's multiplication operator, which makes different conversions according to the context.

As an example, consider an NxN matrix `A` and an N vector `b`. If you give Algae the product `A'*A*b`, it first multiplies `A'*A` which involves an order of N^3 operations. But if you do it as `A'*(A*b)`, there are only an order of N^2 operations. The difference can be huge! With this function, you'd do it as `mult(A';A;b')`. (Note that we have to explicitly convert `b` to a matrix with the right shape.)

solve (*A; b; opts*) Function

This function solves the linear equations `A*x=b` for *x*. Conceptually it returns `inv(A)*b`, except that `solve` is generally much more efficient and accurate. If *A* has already been factored by the `factor` or `chol` functions, then its factors (in the form of a table returned by those functions) may be given in the place of *A*. If the optional argument *opts* contains the member "pos", then the Cholesky method is used to factor *A*.

See also `backsub`, `chol`, and `factor`.

ssi (*A; B; V; eps*) Function

The `ssi` function uses subspace iteration to compute the first few eigenvalues and vectors for a real, symmetric, generalized eigenvalue problem `A*v=B*v*w`. The terms *w* and *v* are the eigenvalues and eigenvectors, respectively.

Subspace iteration is an effective approach for large, sparse problems and for problems for which good estimates of the eigenvectors are available. Otherwise, it sucks.

Both *A* and *B* must be real, symmetric, non-singular matrices. The matrix *V* contains a set of starting vectors, one to a column. A convergence tolerance may be specified with *eps*; 1.0E-6 is used if *eps* is NULL. A table is returned, containing the members "values" and "vectors".

See also `eig`.

svd (*A; opts*) Function

This function computes the singular value decomposition (SVD) of the matrix *A*. In the full case, the SVD is written `A=U*S*V'`, where *S* is a real matrix with the same dimensions as *A* and *U* and *V* are the orthogonal (unitary) left and right singular vectors. The matrix *S* contains the singular values on its diagonal and is zero elsewhere.

When *A* is not square, one or more rows or columns of *S* are known *a priori* to be zero. By default, `svd` then returns a "compact" form of the singular vectors, in which the corresponding columns of *U* or *V* are not included. This behavior may be modified by using the `full` option, as described below.

The argument *opts* is an options table; it may contain the members `novectors` and `full`. (The values of these members are irrelevant; only their existence is signficant.) If *opts* contains the member `novectors`, then only the singular values are computed. If *opts* contains only the member `full`, then the full singular vectors are computed.

The results of `svd` are returned in a table. The member `sigma` contains the vector of singular values, sorted in decreasing order. Unless the `novectors` option is specified, the members `U` and `V` contain the corresponding left and right singular vectors.

This function calls the LAPACK routines DGESVD and ZGESVD.

transform (*A; P*) Function

The `transform` function performs the linear coordinate transformation `P'*A*P`, where *A* is a square matrix. In fact, there are only two differences between the operator expression `P'*A*P` and the function expression `transform(A;P)`:

- The `transform` function results in the most appropriate symmetry. For example if *A* and *P* are both real, and *A* is symmetric, then the result is a symmetric matrix. With the operator expression this would not be the case; the result would be exactly the same (neglecting any roundoff errors) except that it would be marked with general symmetry.

- The `transform` function is considerably faster than the operator expression for some matrix combinations. In particular, it may be two or three times faster when P is dense and A is large, sparse, and either hermitian or real symmetric.

6.5 Numerical Analysis

brent (*f; a; b; tol; param*) Function

This function uses Brent's method to find a single root of the function f; that is, a solution to `f(x)=0`. The ordinates A and B must bracket a root, meaning that `f(A)*f(B)` must be negative. If *param* is given, it is passed as the second argument to the function *f*.

monte (*f; limits; tol*) Function

This function uses the Monte Carlo method to integrate a function. The function values are obtained by calling the function f with one vector argument. The matrix *limits* describes the rectangular region over which the function is integrated. It has two columns and as many rows as f has dimensions. Each row specifies the extremes of the region for the corresponding dimension.

The *tol* argument specifies an error tolerance. (This argument is required: there is no default.) There is no limit to the number of points chosen; `monte` continues to pick points until its one standard deviation error estimate is less than *tol*. Generally, the accuracy increases only as the square root of the number of points evaluated.

The Monte Carlo method provides a simple, easy approach for solving (numerically) even complicated, multi-dimensional integrals. However, it is notoriously inefficient. Each additional digit of accuracy requires 100 times more effort.

Here is a simple example, in which we compute the area of a circle with unit radius. With *domain*, we specify a square centered on the origin within which to integrate. The number of independent variables in the problem (2 in this case) is determined by the number of rows of *domain*. The function f takes as its argument a vector of the independent variables and returns the function value at that point. It will be called repeatedly, with the independent variables spread randomly within the domain. For this problem, f returns 1 if the point is inside the circle, and 0 otherwise.

```
domain = [-1,1; -1,1];
f = function (x) { return norm(x) < 1; };
monte (f; domain; 0.1)?
        3.040
monte (f; domain; 0.01)?
        3.120
monte (f; domain; 0.001)?
        3.142
```

As you can see from the results, the third call to `monte` gave a more accurate result, but it took 10,000 times the effort used in the first call.

By the way, this problem could be solved faster with a one-dimensional integration. Then again, we already know the answer, don't we?

ode4 (*f; t0; tf; x0; y; tol; h; s*) Function

The `ode4` function uses fourth and fifth order Runge-Kutta formulas to solve an initial value problem involving a system of ordinary differential equations. The function *f*, called as `f( t; x )`, is expected to return the first derivative of the state vector *x* with respect to the independent variable *t*. The arguments *t0* and *tf* are the initial and final values of the independent variable, and *x0* is the initial state vector. The argument *y* may be a function, called as `y( t; x )`, that returns a vector of output values. If *y* is NULL, the output vector is the state vector itself.

The optional argument *tol* specifies the desired accuracy; we use 0.001 if *tol* is NULL. An initial stepsize may be suggested with *h*. The *s* argument may be used to provide better control over the accuracy—if *s* is not NULL, then a step is accepted if the estimated error in each state, divided by the corresponding term of *s*, is not greater than *tol*.

The return value is a matrix containing the output history. Each column contains the output from *y* for the particular value of the independent variable given by the column label.

roots (*c*) Function

This function computes the roots of the polynomial having the coefficients given by the vector *c*. If *c* has `n+1` elements, the equation solved (for `x`) is

```
c[1]*x^n + c[2]*x^(n-1) + ... + c[n]*x + c[n+1] = 0
```

A vector with zero elements is returned if the equation has no solution (`roots(1)`, for example). NULL is returned if it has an infinite number of solutions (`roots(0)`, for example). Otherwise, the vector returned contains all of the solutions.

spline (*a; x*) Function

This function computes or evaluates natural cubic spline interpolations for given data. If called with one argument, *a*, a spline for the points in the vector *a* is computed. The elements of *a* give the ordinates; the labels of *a* give the abscissae. If *a* has no numeric labels, then the natural numbers from one to the length of *a* are assumed. In any case, the vector *a* is sorted with respect to its labels; they must be unique.

If `spline` is called with two arguments, *a* and *x*, then the spline *a* (or a spline interpolation of *a*, if *a* is not a spline) is evaluated at the abscissa values given by the vector *x*.

trapz (*y*) Function

This function computes an approximation of the integral of *y* using the trapezoidal method. If *y* has numeric labels, they are used as its abscissae; otherwise, unit spacing is assumed. For the result to make sense, you'll probably want the labels to be in monotonically increasing order.

As an example, use `trapz` to compute the definite integral of `1/x` from 1 to 2.

```
x = linspace (1; 2; 11);
y = label (1/x; x);
trapz (y)
        0.6938
log(2)-log(1)
        0.6931
```

6.6 Basic I/O

close (*file*) Function

The `close` function closes the specified file. The file name *file* must be given exactly as it was when the file was opened. (Whitespace is significant.) Any buffered data is flushed.

digits (*n*) Function

The `digits` function may be used to specify the number of digits that Algae uses when printing real and complex numbers. This applies to the `print` function and to "printing statements" (those with question mark or newline terminators). The argument *n* specifies the number of digits. If *n* is NULL, no action is taken.

The function returns the number of digits being used. By default, Algae uses 4 digits.

Here's an example interaction, with user input preceded by the "`>`" prompt:

```
> digits ()      # the default
        4
> acos (-1)      # pi, to 4 significant digits
        3.142
> digits (20);   # set to 20 digits
> acos (-1)      # pi, to 20 digits (but only accurate to 17)
        3.1415926535897931000
```

This function simply sets the global variable `$digits`, so you can do the same thing by assigning to it directly.

See also `print`.

fread (*file; length; type*) Function

The `fread` function reads data from a binary file into a numeric array. This is a low-level routine, and it requires that you know precisely the format of the data that is being read. To read Algae's own binary files (those written with `put`), see `get`.

The *file* argument (a character scalar) is the file name. As with any file name, if its first character is an exclamation mark then the rest of the string is given to the shell as a filter. If *file* is NULL, then standard input is read.

The *length* argument (an integer scalar) gives the number of items to read. If it's NULL, the entire file is read. Otherwise, it should be a non-negative integer. A vector is returned that contains the data that was read. If `fread` reaches the end of the file before having read *length* elements, then the vector it returns will contain fewer than *length* elements.

The *type* argument is an optional table; its member's names specify the type of data to read and, implicitly, the type of the returned array. (The values of this table's members are inconsequential; only their names are important.) These names may include any of the C language types:

`char` Each is cast to an Algae `integer`.

`int` Same as an Algae `integer`.

`float` Each is cast to an Algae `real`.

`double` Same as an Algae `real`.

The C language type qualifiers are also accepted in *type*:

`long` A qualifier for `int` and `double` types. Both are cast to `real`.

`short` A qualifier for `int`. Casts to `integer`.

`signed` A qualifier for `char` and `int`. Casts to `integer`.

`unsigned` A qualifier for `char` and `int`. An `unsigned char` casts to `integer`, while an `unsigned int` casts to `real`.

Combinations that are disallowed in C (such as `unsigned double`) are disallowed here, too. If *type* is NULL or empty, `double` is assumed.

Besides the C language type qualifiers just described, the following special qualifiers are also accepted:

`big` Specifies the so-called "big endian" byte order.

`little` Specifies the so-called "little endian" byte order.

`ieee` Specifies the IEEE 754 binary floating point format.

fprintf (*file; format; . . .*) Function

The `fprintf` function prints formatted output to *file*. The *format* argument is a character string that contains two types of objects: characters and conversion specifications. Characters are printed directly; conversion specifications cause the next argument to be formatted and printed.

Conversion specifications are introduced using the percent sign '`%`'. Although extensions are planned, the conversion specifications are currently identical to those for the "fprintf" function of the ANSI C language.

If the first character of *file* is an exclamation mark, then the rest of *file* is given to the shell as a filter to which the printed output of `fprintf` is sent. For example, the command `fprintf("!sort";"a\nc\nb\n");close("!sort");` sends "a", "c", and "b", each on a separate line, to the system's sort function, which then presumably sorts and prints it. If *file* is NULL, stdout is used.

See also `message`, `print`, `printf`, `sprintf`, and `put`.

fwrite (*file; v; type*) Function

The `fwrite` function writes binary data to a file from a numeric array. This is a low-level routine, and it requires that you specify precisely the format of the data being written. To write an entity for use in Algae, you'll probably want to use `put` instead.

The *file* argument (a character scalar) is the file name. As with any file name, if its first character is an exclamation mark then the rest of the string is given to the shell as a filter. If *file* is NULL, then the data is written to standard output.

The data to be written is given in the argument *v*, a numeric vector.

The *type* argument is an optional table; its member's names specify the type of data to write. (The values of this table's members are inconsequential; only their names are important.) These names may include any of the C language types and qualifiers, and a few others. See `fread` for details.

message (*fmt; . . .*) Function

The `message` function is used to write error messages to stderr. Its arguments are exactly like `printf`, except that a maximum of 10 arguments is allowed. The message written includes the program and file names as well as the line number.

See also `printf`.

print (*x; file*) Function

The `print` function prints its argument *x* to a file named by *file*. This output goes through the same formatting that Algae uses to print entities to the screen. In particular, the variable `$term_width` controls the width of this output, and the `$digits` variable specifies the number of significant digits printed for real and complex numbers.

If *file* is NULL, then the output is written to stdout.

See also `digits`, `fprintf`, `printf`, `sprintf`, and `put`.

printf (*format; . . .*) Function

The `printf` function prints formatted output to stdout. It is exactly the same as calling `fprintf` with NULL as the first argument.

See also `fprintf`, `message`, `print`, `put`, and `sprintf`.

read (*file*) Function

This function reads a line of text and returns it as a character string. The argument *file* may be a file name or a pipe; see `fprintf` for details. If *file* is NULL, the text is read from standard input.

When `read` reaches the end of file, it returns NULL.

See also `get` and `readnum`.

readnum (*shape; file*) Function

This function reads real numbers from a file named *file*. The vector *shape* specifies the form of the output. If *shape* is NULL or has zero length, a scalar is returned. If *shape* has length one, then a vector with `shape[1]` elements is returned. If *shape* has two elements, then a matrix is returned with `shape[1]` rows and `shape[2]` columns.

The values are read from *file*, or from stdin if *file* is NULL or not given. The '`#`' symbol is special—it and all remaining characters on a line are ignored. If the end of file is reached before the specified number of values is read, then the result is filled out with zeros. After a call to `readnum`, the number of values actually read is contained in the variable `$read`.

For example, let's say you have a file called '`jnk`' that contains the following:

```
# this line is ignored -- numbers like 1.23 are skipped
the first two numbers are 42 and 99.
3.14 is the next one
```

In the following interaction we read the numbers from the file into a matrix:

```
> x = readnum (2,4; "jnk")?
        [         42.00         99.00         3.140         0.000 ]
        [         0.000         0.000         0.000         0.000 ]
> $read?
        3
> readnum (3; "jnk")?
        (        0.000,        0.000,        0.000 )
```

As you can see, anything that doesn't look like a number is simply skipped. The `readnum` function found three numbers and filled in the rest of the matrix with zeros. The global variable `$read` was set to 3 – the number of values actually read.

Subsequent calls to `readnum` with the same file name simply continue reading from the last position in the file. In the last statement of the previous example, no more numbers were found so the vector was filled with zeros. To read again from the start of the file, you must first close the file with the `close` function.

The `readnum` function recognizes integers and floating point numbers. A floating point number consists of an optionally signed integer part, a decimal point, a fraction part, an '`e`' or '`E`', and an optionally signed integer exponent. All of the following examples are recognized as numbers: '`1`', '`-1.2`', '`1e2`', '`-1.e2`', '`.1e2`', '`-1.2e3`', and '`1.2e-3`'.

Fortran users take note! A Fortran double precision number such as '`1.23D4`' will not be read by `readnum` as a single number. The character '`D`' is not recognized as part of a floating point number, so `readline` will read this as the two numbers '`1.23`' and '`4`'.

sprintf (*format; . . .*) Function

The `sprintf` function writes formatted output to a character scalar. It works the same as `printf`, except that the output is returned instead of printed. For example, `x=sprintf("y=%d";1)` puts the string "y=1" in the scalar *x*.

See also `fprintf`, `printf`, and `put`.

tmp_file ( ) Function

The `tmp_file` function creates a temporary file and returns its name. This file will be deleted automatically when Algae exits.

By default, Algae writes temporary files in the directory '`/tmp`'. You can override that with an environment variable called `TMPDIR`. For example, if you use the Bourne shell and put the lines

```
TMPDIR=/usr/tmp
export TMPDIR
```

in your '`~/.profile`' file, then Algae will put its temporary files in the '`/usr/tmp`' directory.

This function not only creates the file, but opens it for output. If you want to read from the temporary file, then you should close it first. (Of course, you'll need to arrange to get something in it, first.) Notice that temporary files are not deleted when closed, but only when Algae exits.

6.7 Entity I/O

get (*file*) Function

The `get` function reads an Algae entity from a binary file named *file*. If no argument is given, standard input is used. Use this function to read files written with `put`. The file name can specify a pipe; see `fprintf` for more details.

NULL is returned if an error occurs in reading the file; otherwise `get` returns the entity it read.

getdyn (*fname*) Function

This function reads matrices from the DYNASTY file *fname*. (DYNASTY is the name of a software package used at Boeing for dynamic analysis.) A table is returned, containing the matrices as members.

The file name can specify a pipe; see `fprintf` for more details.

getmat (*fname*) Function

The `getmat` function reads matrices from a MATLAB binary file named *fname*. It returns the matrices in a table.

The file name can specify a pipe; see `fprintf` for more details.

Notice that `getmat` doesn't assign the matrices to the global symbol table as MATLAB's "load" function does. To do that, you might use something like the following:

```
Load = function (fname)
{
  local (t; n);
  t = getmat (fname);
  for (n in members (t))
  {
    $$.(n) = t.(n);
```

```
    }
  };
```

The `getmat` function reads the MATLAB v.4 "Level-1 MAT-file" format. MATLAB can read and write in this format, but its default for writing is now Level-2. Using the newer format would require us to link to their libraries, meaning that you'd need to actually have MATLAB on that machine. We'll stick with the older format.

One unfortunate consequence of using the older file format is that matrices cannot be written in sparse form. To get around this, convert the matrix to coordinate form (which is dense) within MATLAB and then change it back to sparse form in Algae with the `mksparse` function.

See also `get`, `put`, and `putmat`.

load (*fname*) Function

The `load` function reads a table from a file named *fname* and assigns its members as global variables. This can be used, for example, to restore an Algae session that was saved with the `save` function.

If an error occurs in reading the file, `load` returns 0; otherwise it returns 1.

See also `get`, `put`, and `save`.

put (*x; fname*) Function

The `put` function writes *x* to a binary file named *fname* (or standard output if *fname* is NULL). This binary file can be read with `get`. The file name can specify a pipe; see `fprintf` for more details.

NULL is returned if an error occurs in writing the file; otherwise a 1 is returned.

putdyn (*x; fname*) Function

This function writes the members of the table *x* to a DYNASTY file named *fname*. (DYNASTY is the name of a software package used at Boeing for dynamic analysis.) The file name may specify a pipe; see `fprintf` for more details.

The file is not closed when `putdyn` finishes, so subsequent writes to the same file will append data to the end of it unless you close it first. On some machines, this could be used to build a complete DYNASTY file with several different calls to `putdyn`. The disadvantage, though, is that the DYNASTY file format doesn't allow this on all machines.

putmat (*t; fname*) Function

The `putmat` function writes a table of matrices *t* to a MATLAB binary file named *fname*. The file name can specify a pipe; see `fprintf` for more details.

NULL is returned if an error occurs in writing the file; otherwise a 1 is returned.

NULL members in *t* are skipped; otherwise each member is converted to a matrix if necessary and then written to the file.

The `putmat` function writes in the MATLAB v.4 "Level-1 MAT-file" format. MATLAB can read and write in this format, but its default for writing is now Level-2.

Using the newer format would require us to link to their libraries, meaning that you'd need to actually have MATLAB on that machine. We'll stick with the older format.

One unfortunate consequence of using the older file format is that matrices cannot be written in sparse form. To get around this, use the `exsparse` function to convert the matrix to coordinate form (which is dense) and then change it back to sparse form within MATLAB.

See also `get`, `getmat`, and `put`.

save (*fname*) Function

The `save` function writes all non-NULL global variables as a table to a file named *fname*. This can be used, for example, to save an Algae session that you wish to start again later using the `load` function.

NULL is returned if an error occurs in writing the file; otherwise `save` returns 1.

See also `get`, `load`, and `put`.

6.8 Execution

autosrc (*name; file*) Function

This function creates an *autosrc* function named *name*. When this new function is later called, it replaces itself with a definition that it reads by calling the `src` function with the file name *file*. In this way, you can put off reading a function's definition until it is actually called.

If *file* is `NULL`, then `autosrc` uses the value of *name* for the file name.

For example, consider the following code:

```
autosrc ("yow");
yow ();
```

The first line defines an autosrc function named `yow`. When `yow` is called on the second line, it reads the real definition of `yow` by calling `src` with the file name `"yow"`. It then calls its replacement (with the same arguments, of course).

The current implementation of `autosrc` restricts the number of arguments to 10. (It's an ugly restriction, but if you need that many arguments perhaps you should be using a table, instead.) A zero returned indicates success.

builtin (*file; symbol*) Function

On machines that support dymamic linking, this function creates a new builtin function by linking to a shared object named by *file* and locating the symbol named by *symbol*.

Suppose you have some old Fortran code that computes your chances of being hit by a comet. You can write a C wrapper for it, link them as a shared object, use `builtin` to attach it, and, voila!, it's a builtin function.

Unlike other functions in Algae that deal with files, `builtin` does not accept the special "bang" type file names. (These are file names that begin with an exclamation point and are given to the shell as a command.)

It isn't hard to create shared objects for `builtin`, but currently there's no documentation for it. Perhaps you can find some code to imitate.

cd (*path*) Function

The `cd` function changes the current default directory to that specified by the character scalar *path*. If *path* is NULL, the directory is changed to that given by the `HOME` environment variable.

An exception is raised if an error occurs; otherwise, `cd` returns the integer 1.

eval (*s*) Function

The `eval` function evaluates the string *s* as an expression and returns its value. Variable references, if any, have global scope. For example, consider the following interaction:

```
> x = 0.5;
> str = "sin(x)";
> eval (str)
        0.4794
> x = 1.0;
> eval (str)
        0.8415
```

Although the value of `str` remains `"sin(x)"`, the result of `eval(str)` changes when the value of `x` changes.

Unlike its prominent place in some matrix programming languages, `eval` is rarely necessary in Algae. Most examples of its use are rather contrived. Here's some code which evaluates an expression typed in by the user:

```
x = eval (read ("!echo -n \"expression?  \" 1>&2 ; head -1"))
```

One use for `eval` that may not be quite so contrived involves building a function within a function—something that you can't do directly in Algae. Here's an example:

```
add_const = function (c)
{
  return eval (sprintf ("function (x) { return x+%g; }"; c));
};
```

This could then be used as follows:

```
> ten_more = add_const( 10 );
> ten_more(1)?
        11
```

Probably the most common mistake made with `eval` is giving it a statement instead of an expression. If that's really what you want to do, use the `exec` function instead.

Remember, too, that all variable references are at global scope. For example, the function

```
function( a; b ) { return eval( "a + b" ); }
```

returns the sum of the global variables `a` and `b`; it completely ignores the function's arguments, as they are local variables.

See also `exec`.

exception ( ) Function

This function "raises an exception", meaning that Algae makes an error return. If execution is interactive, control returns to the user. Otherwise, Algae exits.

exec (*s*) Function

The `exec` function causes Algae to parse and execute the Algae statements contained in the character string *s*. Execution begins at global scope, even if `exec` is called from a function. It returns a negative integer if an error occurs, and zero otherwise.

See also `eval` and `source`.

exit (*status*) Function

This function terminates Algae. The optional argument *status* specifies the exit status that Algae returns. On most computer platforms, the exit status is expected to be a small, positive integer. The *status* argument to `exit` should be numeric scalar. If `exit` is called with anything else, Algae returns 0 as its exit status.

A call to `exit` is not mandatory in your Algae programs. Algae terminates when it reaches the end of its input, returning exit status 0. If an error causes Algae to terminate, exit status 1 is returned.

file ( ) Function

The `file` function returns the name of the file from which statements are currently being executed.

get_path (*env; def*) Function

The `get_path` function returns a path list, for use in file searching. The argument *env*, if it isn't NULL, gives the name of an environment variable containing a colon-separated list of paths. The *def* argument gives a default value, in case *env* isn't given or doesn't exist.

See also `search_path` and `src`.

getenv (*ename*) Function

The `getenv` function returns a character scalar that contains the value of the environment variable named by *ename*. If no such variable exists, NULL is returned.

line ( ) Function

The `line` function returns the line number from which it is called.

provide (*feature*) Function

A *feature* is simply a name that stands for a particular collection of functions or other entities. A program announces that a feature is present by calling the `provide` function with *feature*, the name of that feature. This means that the facilities associated with *feature* have been made available for other Algae programs.

Programs that require a particular feature can source the appropriate file with the `require` function.

The argument *feature* must be a character scalar. It is added to the global variable `$features` if it is not already present in that vector. The `provide` function returns *feature*.

require (*feature; filename*) Function

A *feature* is simply a name that stands for a particular collection of functions or other entities. A program that needs the collection may ensure that they are defined by calling `require` with the name of that feature. The use of named features simplifies the task of determining whether required assignments have been made.

The `require` function checks for the feature named *feature* in the current Algae session; if it isn't present, then `require` reads the source file *filename* with `src`. If *filename* is `NULL`, then the value of *feature* is used as the file name.

Before it finishes, the source file read by `require` is expected to call the `provide` function, indicating that the desired feature has been provided. An exception is raised if the source file cannot be read or does not provide the feature named *feature*.

search_path (*fn; suffices; path*) Function

The `search_path` function searches for a file in the directories named in *path*. The file name begins with the string given by *fn*, with one of the suffices given by *suffices*.

source (*fname*) Function

This function causes Algae to parse and execute the file named *fname*. Execution begins at global scope, even if `source` is called from a function. It returns NULL if it has a problem opening *fname*, a negative integer if a parse or run time error occurs, and zero otherwise.

The file is closed after it is read.

See also `exec` and `sources`.

sources (*fname*) Function

The `sources` function causes Algae to parse and execute specified files. The character scalar *fname* is passed through sh(1) to ls(1), so file name generation is available and multiple names may be given. For example,

```
sources( "*.A" );
```

sources all of the files ending with '`.A`' in your working directory. Although you could use the `source` function to do this as

```
source( "!cat *.A" );
```

using `sources` is preferable because it maintains the correct file and line information for the interpreter.

The "pipe" capability normally available to Algae I/O functions is not permitted in `sources`. Command substitution works, though, as demonstrated here:

```
sources( "`find $HOME -name '*.A' -print`" );
```

This example sources every file ending with '`.A`' found in any subdirectory under your home directory.

src (*filename*) Function

The `src` function finds and opens a file of Algae code, executes the statements within it at global scope, and then closes the file. This function, rather than `source` or `sources`, is intended to be the normal way to source files directly. Other functions that may be used to source files are `require` and `autosrc`.

To find the file, `src` looks first for a file named '*filename*`.A`', that is, it appends '`.A`' to the name given by the argument *filename*. If a file with that name exists, it is sourced; otherwise, `src` looks for a file named *filename* with nothing appended, and sources it if it exists.

If *filename* is a relative file name, such as '`foo`' or '`foo/bar.A`', `src` searches for the file using the variable `$src_path`. It appends *filename* to each of the directories listed in the character vector `$src_path` (both with and without the '`.A`'), and sources the first file it finds whose name matches. The current default directory is tried only if it is specified in `$src_path`.

No directory searching is performed if *filename* is an absolute file name. File names that begin with '`/`', '`./`', '`../`', or '`~/`' are absolute. If the file name begins with '`~/`', the tilde is replaced with the value of the `HOME` environment variable.

A file name that begins with the '`!`' character is a pipe; the rest of the file name is given as a command line to the shell, the standard output of which is then read by `src`. For example,

```
src("!m4 foo.A")
```

runs the file '`foo.A`' through the `m4` macro preprocessor first before `src` reads it.

The `$src_path` vector is initialized from the environment variable `ALGAE_SRC_PATH`, if it exists. It may be modified in either of the startup files '`/usr/local/lib/algae/4.3.6/algae.A`' or '`$HOME/.algae`'. (See Section 7.1 [Startup Files], page 93.)

The syntax of `ALGAE_SRC_PATH` is the same as that of `PATH` for the shell; fields are separated by '`:`', and '`.`' is used for the current default directory. To set `ALGAE_SRC_PATH` in the Bourne shell, type something like the following:

```
export ALGAE_SRC_PATH
ALGAE_SRC_PATH=.:~/algae:/usr/local/lib/algae
```

Execution of the Algae code in *filename* begins at global scope. For example, if the source file contains the single statement `x=1`, the assignment is made to the global variable *x* regardless of whether `src` was called from within a function or not.

The `src` function raises an exception if *filename* cannot be found. It returns `NULL` if it cannot be opened, a negative integer if a parse or run-time error occurs, and 0 otherwise.

strip (*f*) Function

The `strip` function removes all file and line information from the given function *f*. The parser ordinarily includes this information so that, for example, run-time error messages can identify the line on which they occurred. If an error occurs in a function that has been stripped, then the error is attributed instead to the calling expression.

This function may also be useful in conjunction with profiling. Any time spent in a call to a stripped function gets charged to the file and line from which it was called.

All members of f are left unchanged as members of the return value.

system (*s*) Function

This function passes the string *s* to the command processor for execution. On a UNIX system, for example, the statement `system("ls")` lists the files in the current directory. If the call succeeds, the exit status is returned; otherwise a –1 is returned.

6.9 Special Tools

npsol (*objective; start; constraints; options; state*) Function

The `npsol` function uses a sequential quadratic programming method to minimize a given objective function subject to given constraints. The domain of the objective function is called the "design space", and the coordinates of the design space are called "design variables".

This function uses the NPSOL package of Fortran routines developed and sold by Stanford University. Your version of Algae might not have the `npsol` builtin function; if not, it probably means that you don't have a license for NPSOL or that there was some problem installing it on your computer.

The `npsol` function may not be called recursively.

The *objective* argument may be either a function or a table. If a function, then it's assumed to take one argument, a vector of design variables, and return the value of the objective at that point. If *objective* is a table, it may have the following members:

`objf` A function that returns the value of the objective function for the vector of design variables given as its first argument. If the member `params` exists in *objective*, then it is passed as a second argument to `objf`.

`objgrd` A function that returns the sensitivities of the objective function; that is, the first derivatives with respect to the design variables. Its arguments are the same as for `objf`.

`params` This member, if it exists, is passed as the second argument to `objf` and `objgrd`, providing a means for passing additional parameters to those functions.

Only the `objf` member is required. If the objective derivatives are not provided, you must ensure that `derivative_level` is set accordingly.

The *start* argument is a vector that specifies the starting point.

The *constraints* argument is a table that may contain any of all of the following members:

`side_constraints`

A matrix that specifies upper and lower constraints on values of the design variables. This matrix has two columns, the first column giving lower bounds and the second column giving upper bounds. If this member is not given, then a default is provided using 1.0E10 as "infinity". In that case, you should be aware that NPSOL will take these values literally if you also use the "infinite_bound" option (in *options*) with a value greater than 1.0E10.

`linear_constraints`
A table that may contain any or all of the following members:

`coefficients`
A matrix of the linear constraint coefficients. The i-th row contains the coefficients for the i-th linear constraint.

`bounds` A matrix, like `side_constraints`, that gives the upper and lower bounds for the linear constraints. If not given, defaults are provided as with `side_constraints`.

`nonlinear_constraints`
A table that may contain any or all of the following members:

`values` Either a function that computes the vector of nonlinear constraint function values at the given point, or a table in which member `conf` is a function returning the values and `congrd` is a function returning the gradients. The constraint gradients, if provided, should be given as a matrix in which the rows correspond to the constraints and the columns correspond to the design variables. As in *objectives*, a member called `params` may also be included.

`bounds` A matrix, like `side_constraints`, that gives the upper and lower bounds for the nonlinear constraints. If not given, defaults are provided as with `side_constraints`.

If the constraint derivatives are not provided, you must ensure that `derivative_level` is set accordingly.

The *options* argument is a table that may contain any of the valid NPSOL options. For example, `{ hessian = "yes" }` specifies the corresponding option in NPSOL. Use an underscore instead of blanks between words, as in `{ major_print_level = 1 }`. The valid options are:

`central_difference_interval`
If the algorithm switches to central differences because the forward-difference approximation is not sufficiently accurate, this value is used as the difference interval for every variable.

`cold_start`
`warm_start`
These flags control the specification of the initial working set in both the procedure for finding a feasible point for the linear constraints and bounds, and in the first QP subproblem thereafter. With a `cold_start`, the first working set is chosen by `npsol` based on the values of the variables and constraints at the initial point. Broadly speaking, the initial working set will include equality constraints and bounds or inequality constraints that violate or "nearly" satisfy their bounds within `crash_tolerance`. With a `warm_start`, the user must provide the `state` table. A warm start will be

advantageous if a good estimate of the initial working set is available—for example, when `npsol` is called repeatedly to solve related problems.

`crash_tolerance`

This value is used in conjunction with the optional parameter `cold_start`, which is the default. When making a cold start, the QP algorithm in `npsol` must select an initial working set. The initial working set will include, if possible, bounds or general inequality constraints that lie within `crash_tolerance` of their bounds. The default value is 0.01. If `crash_tolerance` is less than zero or greater than one, the default value is used.

`derivative_level`

This value indicates which objective and constraint derivatives are provided by the user. The choices are as follows:

`0` Neither objective nor constraint derivatives are provided.

`1` The objective derivatives are provided by the `objgrd` function in argument *objective*. Constraint derivatives are not provided.

`2` The nonlinear constraint derivatives are provided by the `congrd` function in argument *constraints*. Objective derivatives are not provided.

`3` The objective derivatives are provided by the `objgrd` function in argument *objective*, and the nonlinear constraint derivatives are provided by the `congrd` function in argument *constraints*.

The value 3 is the default, and should be used whenever possible, since `npsol` is more reliable and will usually be more efficient when all derivatives are exact.

If `derivative_level` is 0 or 2, `npsol` will estimate the objective gradients using finite differences. The computation of finite-difference approximations usually increases the total run time, since a call to the objective function is required for each constraint. Furthermore, less accuracy can be attained in the solution.

If `derivative_level` is 0 or 1, `npsol` will approximate the constraint gradients. One call to the constraint function is required for each variable. At times, central differences are used rather than forward differences, in which case twice as many calls to the objective and constraint functions are needed. The switch to central differences is not under the user's control.

`difference_interval`
: This value defines an interval used to estimate gradients by finite differences in the following circumstances: (1) for verifying the objective and constraint gradients, and (2) for estimating the objective or constraint derivatives. If a difference interval is not specified by the user, a finite difference interval will be computed automatically for each variable by a procedure that requires up to six calls of the objective and constraint functions for each component. This option is recommended if the function is badly scaled or the user wishes to have `npsol` determine constant elements in the objective and constraint gradients.

`feasibility_tolerance`
: This value defines the maximum acceptable absolute violations in linear and nonlinear constraints at a "feasible" point; i.e., a constraint is considered satisfied if its violation does not exceed this value. The default value is the square root of machine precision.

`function_precision`
: This value is intended to be a measure of the accuracy with which the objective and constraint functions can be computed. It acts as a relative precision when the magnitude is large, and as an absolute precision when the magnitude is small. For example, if the objective function is typically on the order of 1000 and the first 6 digits are known to be correct, then `function_precision` should be set to 1.0E–6. In contrast, if the objective function is typically on the order of 1.0E–4 and the first 6 digits are known to be correct, then `function_precision` should be set to 1.0E–10. The choice of `function_precision` can be quite complicated for badly scaled problems. The default value is machine precision raised to the 0.9 power; this is appropriate for most simple functions that are computed with full accuracy. However, when the accuracy of the computed function values is known to be significantly worse than full precision, then `function_precision` should be large enough that `npsol` will not attempt to distinguish between function values that differ by less than the error inherent in the calculation.

`hessian`
: This option must be either `"yes"` or `"no"`. (The default is `"no"`). This option controls the contents of the `R` matrix in *state*. The user should set `hessian` to `"yes"` if the `state` table returned is to be used for a subsequent warm start.

`infinite_bound_size`
: If this value is greater than zero, it defines the "infinite" bound in the definition of the problem constraints. Any upper

bound greater than or equal to this value will be regarded as positive infinity. Any lower bound less than or equal to the negative of this value will be regarded as negative infinity. The default value is 1.0E10.

`infinite_step_size`
: This value specifies the magnitude of the change in variables that is treated as a step to an unbounded solution. If the change in any variable during an iteration would exceed the value of `infinite_step_size`, the objective function is considered to be unbounded below in the feasible region. The default is the greater of `infinite_bound_size` and 1.0E10.

`iteration_limit`
: This value specifies the maximum number of major iterations allowed before termination.

`linear_feasibility_tolerance`
`nonlinear_feasibility_tolerance`
: These values define the maximum acceptable absolute violations in linear and nonlinear constraints at a "feasible" point. A constraint is considered satisfied if its violation does not exceed this value. The default value for both options is the square root of machine precision.

On entry to `npsol`, an iterative procedure is executed in order to find a point that satisfies the linear constraints and bounds on the variables to within `linear_feasibility_tolerance`. All subsequent iterates will satisfy the linear constraints to within the same tolerance, unless this value is comparable to the finite difference interval.

For nonlinear constraints, `nonlinear_feasibility_tolerance` defines the largest constraint violation that is acceptable at an optimal point. Since nonlinear constraints are generally not satisfied until the final iterate, this value acts as a partial termination criterion for the iterative sequence.

These tolerances should reflect the precision of the corresponding constraints. for example, if the variables and the coefficients in the linear constraints are of order unity, and the latter are correct to about 6 decimal digits, it would be appropriate to specify `linear_feasibility_tolerance` as 1.0E–6.

`linesearch_tolerance`
: This value controls the accuracy with which a step taken during each iteration approximates a minimum of the merit function along the search direction. The smaller the value, the more accurate the line search. The default value is 0.9;

it requests an inaccurate search that is appropriate for most problems, particularly those with any nonlinear constraints.

If there are non nonlinear constraints, a more accurate search may be appropriate when it is desirable to reduce the number of major iterations—for example, if the objective function is cheap to evaluate or if the gradients are not specified.

`list` If this flag is given, all of the options and their values are printed.

`print_level`
: This value controls the amount of printed output produced by the major iterations of `npsol`. The levels are as follows:

  `0` No output. This is the default.

  `1` The final solution only.

  `5` One line of output for each major iteration, but no final solution.

  `10` The final solution and one line of output for each iteration.

  `20` At each major iteration, the objective function, the Euclidean norm of the nonlinear constraint violations, the value of the nonlinear constraints, the values of the linear constraints, and the current values of the variables.

`minor_iteration_limit`
: This value specifies the maximum number of iterations for the optimality phase of each QP subproblem.

`minor_print_level`
: This value controls the amount of printout produced by the minor iterations of `npsol`. The levels are as follows:

  `0` No output. This is the default.

  `1` The final QP solution.

  `5` One line of output for each minor iteration, but no final QP solution.

  `10` The final QP solution and one line of output for each minor iteration.

  `20` At each minor iteration, the current estimates of the QP multipliers, the current estimate of the QP search direction, the QP constraint values, and the status of each QP constraint.

`optimality_tolerance`
: This value specifies the accuracy to which the user wishes the final iterate to approximate a solution of the problem.

Broadly speaking, it indicates the number of correct figures desired in the objective function at the solution. For example, if `optimality_tolerance` is 1.0E–6 and `npsol` terminates successfully, the final value of the objective function should have approximately 6 correct figures. The `npsol` function will terminate successfully if the iterative sequence is judged to have converged and the final point satisfies the first-order optimality conditions.

`verify_level`

This value refers to finite-difference checks on the gradients computed by the user function `objgrd` and `congrd`. It may take on one of the following values:

`-1` No checks are made.

`0` Perform only a very inexpensive check, involving one call to the objective function and one call to the constraint function.

`1` Perform a reliable but expensive check on the objective gradients.

`2` Perform a reliable but expensive check on the constraint gradients.

`3` Perform reliable but expensive checks on both the objective and the constraint gradients.

These checks are recommended whenever a new function routine is being developed.

The *state* argument is used only when the NPSOL "warm_start" option is requested. It may be obtained from the return of a previous call to `npsol`. Its members are:

`ISTATE` A vector describing the status of the constraints.

`CLAMDA` The Lagrange multiplier estimates for the nonlinear constraints.

`R` The factored Hessian of the Lagrangian function.

The `npsol` function returns a table containing the following members:

`objective`

The value of the objective function at the final iterate.

`solution` The solution vector (or final estimate) for the problem.

`inform` The INFORM value (success/error indicator) returned by NPSOL. The possibilities are:

`0` The iterates have converged to a point that satisfies the first-order optimality conditions to the accuracy requested by the optional parameter optimality_tolerance, i.e., the projected gradient and active constraint residuals are negligible. (Success.)

1 The final iterate satisfies the first-order optimality conditions to the accuracy requested, but the sequence of iterates has not yet converged. NPSOL was terminated because no further improvement could be made in the merit function. (Probable success.)

2 No feasible point could be found for the linear constraints and bounds. The problem has no feasible solution.

3 No feasible point could be found for the nonlinear constraints. The problem may have no feasible solution.

4 The limiting number of iterations, determined by the optional parameter major_iteration_limit, has been reached.

6 The current point does not satisfy the first-order optimality conditions, and no improved point for the merit function could be found during the final line search.

7 The user-provided derivatives of the objective function and/or nonlinear constraints appear to be incorrect.

9 An input parameter is invalid.

iter The number of major iterations performed.

state The state table described above.

plot (. . .) Function

The plot, splot, replot, and unplot functions provide an interface to the gnuplot program for plotting data. Use plot for plotting lines in two or three dimensions; splot is for plotting surfaces in three dimensions.

To use these plotting functions effectively, you will probably need some familiarity with the gnuplot commands. For example, to add a title to an existing plot, you'd type something like `replot("set title 'Good Stuff'")`. Read the gnuplot manual or its on-line help for more information.

The plot function accepts as many as three arguments. If either or both of the first two arguments are (or can be converted to) character vectors, then their elements are sent, each on a separate line, to gnuplot as commands. For example, `plot("set term X11")` causes gnuplot to generate X11 output. Even commands that request information from gnuplot, like `plot("show term")` are acceptable (although you might temporarily lose sight of Algae's prompt). Don't use the "exit" or "quit" commands of gnuplot—they cause gnuplot to exit without Algae knowing about it.

If more than one argument is given to `plot` and the last one is an integer scalar, then it is taken to be an identifier for the plot process. This allows you to have more than one plot open at a time. If no identifier is given, then the active plot is killed and replaced by a new one with the same identifier. The "active" plot is the one last referenced, or 1 if there are no plots open. The `replot` function may be used to modify an open plot.

Any other arguments given to `plot` are taken to be data to be plotted. Vectors are plotted using their element values as ordinates and their labels for the abscissae. Matrices with 2 columns are plotted using the second column for ordinates and the first column for abscissae. A matrix with 3 columns is plotted as a curve in three dimensions, with the third column specifying the ordinates. Surface plots are obtained using the `splot` function.

If the data given to `plot` is a vector, but has no labels, then the element indices are used instead of the labels.

Multiple curves may be shown on the same plot, either by giving `plot` two data arguments or (better yet) supplying the data in a table. When data is given in a table (even if it contains only one member), the member name is used in the plot legend. Otherwise, `plot` uses the names "y_1" and "y_2".

If the data is given in a table, any members that are tables themselves are given to gnuplot in a single data file. This means that the same line and point type is used. For example, if `x` and `y` are vectors to be plotted, then `plot({x;y})` plots the two, each with a different line and point type and described by name in the legend. On the other hand, `plot({z={x;y}})` plots them with the same line and point type and names the combination "z" in the legend.

If `plot` is called with no arguments, it returns a vector of the open plot identifiers. This vector is sorted from most to least recently referenced, so the "active" plot is first.

Here's an example that simulates the Van der Pol system and plots the results:

```
xdot = function( t; x ) {
    return x[1]*(1-x[2]^2)-x[2], x[1];
}
x = ode4( xdot; 0; 20; 0,.25 );
plot( "set data style lines"; { x1=x[1;]; x2=x[2;] } );
```

See also `splot`, `replot`, and `unplot`.

replot (. . .) Function

The `replot` function is used to modify or redisplay plots created with either the `plot` or `splot` functions. It takes at most 2 arguments. If the first argument has character type, it is passed to gnuplot as commands.

The last argument is the optional plot identifier. If not given, the active plot is used by default.

See also `plot`, `splot`, and `unplot`.

splot (. . .) Function

The `splot` function is used to plot surfaces in three dimensions. Except for the form of the data, its input is identical to that of the `plot` function. The data is specifed as a matrix of ordinates. The labels of the matrix, or the corresponding indices if the labels don't exist, are used for the abscissae.

See also `plot`, `replot`, and `unplot`.

umin (*objective; start; options*) Function

The `umin` function performs unconstrained minimization using the Nelder-Mead direct search method. It does not have the features or sophistication of `npsol`, but it works well for some cases—particularly when the objective surface is not smooth.

The *objective* argument is a function that takes one or two arguments and returns the value of the objective at that point. The first argument is a vector of design variables. The second argument, which is optional, is passed unchanged from the *params* member of the *options* argument to `umin` itself.

The *start* argument is an integer or real vector that specifies the starting point.

The *options* augument may be either NULL or a table containing convergence and other specifications. The meaningful options are:

`bound` The minimum value permitted for the objective function. The default is –1e32.

`display` A function called at each "successful" iteration with the current design point as its only argument. This may be used, for example, to plot the design as it changes.

`evals` The maximum number of objective function evaluations permitted. The default value is 1000.

`iter` The maximum number of iterations permitted. The default value is 100.

`params` The value of this member is passed as the second argument to the objective function.

`right` If this member exists, then the initial simplex has right angles. By default, the initial simplex has sides of equal length.

`size` The initial size of the simplex. By default, this is the greater of 1 and the infinity norm of the starting vector.

`tol` The relative size of the simplex, below which the iteration is taken to have converged. The tolerance is 1e–6 by default.

`verbose` If this member exists, then information is printed at each "successful" iteration.

The return value from `umin` is a table with the following members:

`evals` The number of objective function evaluations.

`inform` A return code specifying success or failure:

`0` success

`1`	failure, objective bound reached
`2`	failure, function evaluation limit exceeded
`3`	failure, iteration limit exceeded

`iter` The number of iterations performed.

`msg` A message corresponding to the inform code.

`obj` The final objective value.

`sol` The final design point.

This code is based on `nmsmax`, a MATLAB function by Nick Higham.

See also `npsol`.

unplot (*id*) Function

The `unplot` function is used to terminate plots created with the `plot` or `splot` functions. The integer scalar or vector argument *id* specifies the identifiers of the plots to be terminated. If no *id* is given, the active plot is terminated. You can terminate all open plots with `unplot(plot())`.

See also `plot`, `replot`, and `splot`.

6.10 Miscellaneous

all (*x*) Function

The `all` function evaluates the "truth" of its argument *x* in the same way that the function `test` does, except that vectors and matrices are "true" only if all of their elements are "true". For example, `all(1,0)` returns 0 while `test(1,0)` returns 1. If *x* has no elements, `all` returns 0.

See also `test` and `equal`.

atof (*s*) Function

The `atof` function converts character strings to real numbers. The argument *s* may be a scalar, vector, or matrix. The function reads only up to the first unrecognized character of each string, and ignores anything that remains. If the first character of a string is unrecognized, then the value is taken to be zero.

char (*v*) Function

The `char` function converts the vector *v* into a character string; each element of *v* contributes a single character according to its ASCII value. For example, `char(65,66,67)` returns the string `"ABC"`.

If an element of *v* is less than 0 or greater than 255, then it will "wrap" (modulo 256). Thus `char(65)` and `char(65+256)` both return the string `"A"`.

Algae's character strings are terminated with a NUL (0) character. For the `char` function, that means that if an element of *v* is zero (or a multiple of 256) then the string is terminated at that point. For example, `char(65,0,66)` yields the string `"A"`.

class (*x*) Function

The `class` function returns a character string (such as "scalar" or "table") that describes the class of *x*.

equal (*a; b*) Function

This function tests *a* and *b* for equality. For vectors and matrices, it returns true (1) only if every pair of corresponding elements is equal, and false (0) otherwise. For all other classes, this function returns the same value as the expression `a==b` would.

See also `test`.

members (*e*) Function

This function returns a vector containing the names of the members of *e*.

info (*topic*) Function

This function starts up an interactive browser for Algae's documentation. The optional argument *topic* (a character scalar) takes you directly to that topic. For example, `info("sort")` takes you directly to the description of the `sort` function.

If possible, `info` uses an X-based html browser (like netscape or mosaic). Otherwise, a character-based html browser (like lynx) will be tried. As a final resort, the GNU Info browser is called.

The names of the available browsers are assigned by the startup code (see Section 7.1 [Startup Files], page 93) as members `xhtml`, `html`, and `info` of the global table `$programs`. To prevent Algae from using a particular browser, simply set the appropriate member of `$programs` to a zero-length string. If you prefer to use Info, for example, you could simply put the line

```
$programs.xhtml = $programs.html = "";
```

in your '`.algae`' file.

prof (*infile; outfile; threshold*) Function

The `prof` function reads an '`algae.out`' file produced by Algae's execution profiler (the '`-p`' option), and summarizes it by file and by line number. The *infile* argument specifies the file name. *outfile* specifies the output file; if it's NULL, stdout is used.

The *threshold* argument may be used to truncate the summary listings. The truncation occurs after enough entries have been printed to account for *threshold* percent of the total number of hits. If *threshold* is NULL, no truncation is performed.

Below is an example of `prof` output, obtained with the command `prof("algae.out";;50);`.

```
Algae execution profile listing.

628 total hits

--- by file ---

     hits   % hits    cum %   file
```

```
        201    32.01    32.01   lqrtest.A
        127    20.22    52.23   /usr/local/lib/algae/ode.A

 --- by line ---

       hits   % hits    cum %   line   file
        130    20.70    20.70     88   lqrtest.A
         42     6.69    27.39      1   /usr/local/lib/algae/plot.A
         40     6.37    33.76     38   lqrtest.A
         21     3.34    37.10      1   /usr/local/lib/algae/ode.A
         17     2.71    39.81     44   /usr/local/lib/algae/ode.A
         15     2.39    42.20     42   /usr/local/lib/algae/ode.A
         15     2.39    44.59     41   /usr/local/lib/algae/ode.A
         14     2.23    46.82      1   /usr/local/lib/algae/ode4.A
         12     1.91    48.73      1   /usr/local/lib/algae/spline.A
         10     1.59    50.32      1   lqrtest.A
```

First, notice that the number of hits counted in this example was only 628. Four digits of precision in the statistics are clearly unjustifed with such a small sample. The first column reports the number of profiler hits counted toward each particular file or line number. The second column restates that as a percentage of the total number of hits. The third column gives the cumulative percentage.

All profiler hits that occur while Algae is parsing a file are counted toward its first line. This is normally insignificant compared to execution, but it explains why line 1 shows up so often in the "by line" report in the example above.

No files are closed in `prof`. Normally, this means that you must close them yourself if `prof` is to be called more than once.

show (*e*) Function

This function prints information about the entity *e* and its members.

split (*s; w*) Function

The `split` function takes a character scalar argument *s*, splits it into tokens, and returns the tokens in a character vector. Each token is delimited by one or more characters from *w*. For example, `split("/bin:/usr/bin";":")` returns the vector

```
( "/bin"    , "/usr/bin" )
```

If *w* is NULL, the string `" \t\n"` is used. Thus `split("This is a test.")` returns the vector

```
( "This" , "is"   , "a"    , "test." )
```

string (*e*) Function

The `string` function converts its argument *e* to character type. If *e* is NULL, a zero-length character scalar is returned. For example, `string(1/(1:3))` returns the vector

```
( "1"        , "0.5"      , "0.333333" )
```

Currently, the output format cannot be changed.

substr (*c; start; length*) Function

The `substr` function returns the substring of the character string *c*, beginning at index *start*, with length *length*. The integer *start* must be greater than zero; if it exceeds the length of *c*, then a zero-length string is returned. If *length* is NULL, then all of the remaining characters of *c* are returned; otherwise, *length* must not be negative.

See also `dice`, `split`, and `string`.

test (*e*) Function

The `test` function evaluates the "truth" of *e* in the same way that Algae's `if` statement would, returning 1 for true and 0 for false. The following entities are false:

- NULL.
- Numeric arrays in which every element is zero.
- Character arrays in which every element is `""`.
- Vectors and matrices with no elements.
- Tables with no members.

All others are true. Notice that if *e* is an array (`scalar`, `vector`, or `matrix`), then `test` returns true if any element of *e* is nonzero (or has nonzero length, for character type).

An example of the use of `test` is Algae's `equal` function, which is written as

```
equal = function( a; b ) { return !test( a != b ); }
```

If `a` and `b` are both matrices, then `a!=b` returns a matrix that is all zeros only if every element pair is equal. In that case, `test` returns 0 and `!` changes that to 1.

See also `equal` and `all`.

time ( ) Function

The `time` function returns the number of seconds of user time that the current process has used. On most machines, its precision is not much more than 1/10 of a second.

tolower (*s*) Function

The `tolower` function converts strings to lower case. Every letter is converted as appropriate for the current locale. The argument S may be a character scalar, vector, or matrix.

See also `toupper`.

toupper (*s*) Function

The `toupper` function converts strings to upper case. Every letter is converted as appropriate for the current locale. The argument S may be a character scalar, vector, or matrix.

See also `tolower`.

what ( ) Function

The `what` function returns a table containing all of the global functions. See also `who`.

who (*opt*) Function

The `who` function returns a table containing all global variables that are neither functions nor NULL. Variables with names that begin with '`$`' are excluded, unless the optional argument *opt* equals `"$"`. See also `what`.

7 Running Algae

The Algae interpreter assembles your Algae statements into its own opcodes and then executes them. When you are working interactively, it does this one statement (or one block of statements) at a time. When its input comes from a file, the entire file is assembled before execution of it begins.

7.1 Startup Files

When Algae begins execution, it normally tries to read and execute two startup files. The first one it tries is the file in which it expects some of its standard functions to be defined. A default name for this file is given when Algae is compiled—usually it's something like '`/usr/local/lib/algae/4.3.6/algae.A`'. That name can be overridden with an environment variable called `ALGAE_RC0`.

Without this file, some of Algae's standard functions will be missing. For this reason, Algae will emit an error message and quit if it can't find the file. (That is, unless the '`-S`' option is specified on the command line.)

After Algae reads its standard functions, it looks for the file '`.algae`' in your home directory and executes it if it's there. This file's name can be overridden with the `ALGAE_RC1` environment variable. Execution of this file is inhibited by the '`-s`' command line option. No startup files at all are read if the '`-S`' option is given.

7.2 The Command Line

Algae supports command line arguments to request various actions. Arguments starting with '`-`' are *options*. Other arguments specify files to execute.

Option flags always begin with a hyphen. Algae supports both traditional single-letter options and mnemonic long option names. Long option names are indicated with '`--`' instead of '`-`'. Abbreviations for option names are allowed as long as they are unique.

Options which change Algae's behavior take effect before any files are executed. The order of the arguments is unimportant, with the following two exceptions:

1. Any input scripts given (with the '`-e`' or '`--script`' option) are executed in the order that they appear on the command line. This occurs after the startup files are executed and before any other files are executed.
2. Any input files named on the command line are executed in the order in which they appear.

Below is a list of options accepted by Algae. Both the short and long option names are indicated.

'`-D`'
'`--disassemble`'
: This option turns on the disassembler, which prints Algae's opcodes to stderr in a form like assembly language. You'll probably want to combine this with the '`-S`' option; otherwise, you'll get more than 800 lines of opcodes from the standard functions.

'`-d` *i*'
'`--debug` *i*'
: This option sets the debug level to *i*, where *i* is an integer. It is normally of use only for debugging the Algae implementation.

'`-e` *commands*'
'`--script` *commands*'
: This option allows you to provide a script for Algae to execute from the command line, rather than from a file or from standard input. Such a script is executed after any startup files but before any other files are executed. Any number of '`-e`' (or '`--script`') options may be given, and the specified scripts are executed in the order in which they appear on the command line.

'`-h`'
'`--help`' Print a brief description of the command line arguments.

'`-i`'
'`--interactive`'
: This option causes Algae to use interactive mode when reading from the standard input device "stdin". Without this option, Algae uses interactive mode only when its input appears to be from a terminal. In interactive mode, input is parsed a line at a time and exceptions do not cause Algae to exit.

'`-n`'
'`--nowhite`'
: This option changes the way that scalars are displayed. The printing statements (that is, statements that are terminated by either a newline or a '`?`' character) normally precede the scalar's value with a tab, and follow it with a newline. With this option set, neither the tab nor the newline is printed. (For character scalars, the quotation marks are also omitted.) This option affects the printing statements only, and has no effect on the functions like `printf`.

'`-p`'
'`--profile`'
: This option enables execution profiling—a means for determining the execution time characteristics of your Algae program. When profiling is enabled, the profiler periodically interrupts Algae and records the line that is currently being executed. When Algae exits, it records this data in the file '`algae.out`' in the current directory.

  Use the `prof` function to read the '`algae.out`' file and summarize it by file and by line number.

  You may wish to use the `strip` function in conjunction with profiling. Since `strip` removes the line and file information from a function, any time spent in a call to that function gets charged to the line from which it was called.

  We've encountered several systems on which operating system bugs prevent the execution profiler from working correctly. These include a DECstation and a Titan, both with MIPS architectures. If you have such a system, then Algae should have been installed with the execution profiler disabled. In that case, you'll get an error message if you try to use the '`-p`' option.

'-R'
'--restrict'
: This option causes Algae to run in restricted mode. The `system` function is disabled, as are "pipes" (i.e., file names that begin with the '!' character).

  Restricted mode should be used whenever you deal with untrusted Algae code. Without it, a malicious provider of such code could cause major damage.

'-r'
'--noreadline'
: If the GNU Readline facility is available, Algae normally uses it for interactive command line editing and history. The '-r' option forces Algae to skip Readline processing. Readline's editing commands are similar to emacs: *C-f* forward, *C-b* backward, *C-p* up, and *C-n* down. You can change to `vi` style by typing *M-C-j*.

'-S'
'--nostartup'
: If this option is given, none of Algae's startup files are read. This means that many of the standard functions will be unavailable.

'-s'
'--norc'
: This option skips reading the user's startup file.

'-V'
'--version'
: This option prints version and date information.

'-x'
'--nostdin'
: This option causes Algae not to read stdin by default when no file names are given on the command line.

Any file names given on the command line are executed as input to Algae. If no file names are given (and the `-x` option is not present), input comes from stdin. You can specify stdin explicitly with a single hyphen, so a command line like

```
algae init.A -
```

has Algae execute '`init.A`' first and then read from standard input.

7.3 Errors

Several types of errors may be encountered when running Algae. The two most common are *parse* errors and *run time* errors. Parse errors occur while Algae is parsing its input statements. Run time errors occur while Algae is executing its code. In both cases, Algae prints a file name and line number associated with the error.

7.4 The algae Prompt

When executing interactively, Algae displays the primary prompt when it is ready to read a command, and the secondary prompt when it needs more input to complete a command.

You can customize the prompt simply by assigning a character vector to the global variable `$prompt`. Its first two elements specify the primary and secondary prompts. By default, `$prompt` is `( "> ", " " )`. Assigning something other than a character vector to `$prompt` is not an error—Algae just won't give you a prompt. (Wouldn't it be fun to accept a function? Hmm, maybe someday.)

If the GNU Readline facility is available (that is, linked with Algae during installation), Algae normally uses it for interactive command line editing and history. Readline's editing commands are similar to emacs: *C-f* forward, *C-b* backward, *C-p* up, and *C-n* down. You can change to `vi` style by typing *M-C-j*. The '`-r`' command line option forces Algae to skip Readline processing.

7.5 Signals

If Algae receives an interrupt signal (because you pressed a *C-c* on the keyboard, for example), it stops what it's doing and returns to the prompt. (If it isn't running interactively, it simply exits.) On Unix systems, you can also use the `kill` command to send a signal to a process.

The implementation of interrupt signal handling in Algae is necessarily a compromise. We want it to respond promptly, but not at the expense of our performance. As a result, it sometimes happens that Algae does not respond promptly to an interrupt signal. (If you find a case like this; please report it. Sometimes it's the result of an oversight.)

Sending Algae a quit signal causes an immediate, clean exit. There are various other signals which will cause Algae to terminate, but this is the one to use if you want your files closed properly, etc.

8 Projects

Below is a list of improvements that we'd like to make to Algae. These items run the gamut from specific changes that would take only an hour or so to implement to vague, half-baked ideas that might take months. The items are in no particular order—we haven't identified priorities on most of them. We'd be very happy to have some help.

- The `atan` and `atanh` functions need improvement for complex arguments. They are based on simple formulas, but better algorithms are available that are faster, more accurate, and less prone to overflow.
- In the old days, getting Readline to do context-based completion (as on variable names) was nearly impossible. This may have changed – we ought to take a new look at it.
- The `printf` and related functions should be given additional capabilities to accept array arguments.
- I'm interested in exploring a better arrangement for the help system. Users should be able to tie in documentation for their own code.
- With the `builtin` function, Algae has dynamic linking capability. To make this useful, though, we have to document the interface and reorganize the header files.
- Wouldn't a GTK+ (`http://www.gtk.org/`) interface be nice?
- We need to complete the `fread` and `fwrite` implementation, and we'll probably want an `fseek` function as well.
- Symbolic debugging of Algae code would be very helpful.

9 Bugs

Your bug reports play an essential role in making Algae reliable. By reporting a bug, you may or may not get a timely solution to your problem. Either way, bug reports help us to make the next version of Algae better. In addition, your comments or criticisms on Algae or the Algae language are also welcomed.

The Algae interpreter is incomplete. There are several operations that it should be able to perform but that have not yet been implemented. These cases elicit a "detour" message from Algae. Since they're not bugs, you don't need to tell us about them. Still, complaining about them might get them fixed.

9.1 Reporting Bugs

In order for a bug report to serve its purpose, you must include the information we need to fix it. As the GNU people say, "report all the facts". But never mind Joe Friday—the more information the better. It usually doesn't pay to explore the "envelope" of the bug; that is, changes to the input that affect it. Providing a simple example is the best way to get a bug fixed.

You should include the following information with your bug report:

- The version of Algae. You can get this by running it with the '`-V`' option.
- A complete input file that will reproduce the bug. A single statement is generally not sufficient.
- The type of machine you are using, and the operating system name and version number.
- A description of what behavior you observe that you believe is incorrect.

Send bug reports to:

```
ksh@sideslip.org
```

or, as a last resort, mail them to:

```
Algae Bugs
Attn: Scott Hunziker
The Boeing Company
P.O. Box 3707, MC 8K-26
Seattle, WA  98124-2207
```

9.2 Reported Bugs

Below is a list of bugs that are known to exist in the current version of Algae and are waiting to be fixed.

1 Recurse deep enough, and Algae dumps core. For example, the expression

```
function () { return self(); } ()
```

will not return gracefully. (I just tried it, and I had to reboot my machine!) It usually takes more than 500 levels of recursion to hit this bug, though, so you'll probably only encounter it with runaway functions.

16 We have problems when a parse error occurs in a recursive call to the parser. Local variables created in the recursive call are left in an invalid state. This happens, for example, if you call the `source` function from an interactive session and encounter a parse error in the process.

22 Every time Algae parses a file, it leaks the memory in which the name of that file is stored. The names are kept around because user functions refer to them. Instead, we should arrange for the names to be stored with each user function so that they get deleted when the function is deleted.

23 When Algae reads a binary file that contains a user function, it allocates memory for every string constant in that function. If you then delete that function, this memory is leaked.

24 We use the texi2html program to make Algae's HTML documentation from the texinfo source. Among its numerous deficiencies, it sometimes splits an anchor from the text with which it is supposed to be associated. For example (from algae-3.5.0), the anchor for the section named "Running Algae" is put at the bottom of the file '`algae_6.html`' instead of at the top of '`algae_7.html`' where it belongs. For this reason, the Algae `info` function may not take you to the right place in the document.

Concept Index

If you can't find what you're looking for here, please send us a bug report. We'll try to include an entry for it in the next release of this manual.

!

! (not operator) 6, 24
!= (inequality operator) 6, 25

#

(begin comment) 29

$

$$ 18

%

% (modulus operator) 25
%= (assignment operator) 27

&

& (and operator) 6, 26
&& (and operator) 6, 26

'

' (conjugate transpose operator) 23

(

() function calls 22

*

* (multiplication operator) 24
*= (assignment operator) 27

,

, (append operator) 4, 27

-

– (negation operator) 24
– (subtraction operator) 25
–= (assignment operator) 27

.

. (member reference operator) 23

/

/ (division operator) 25
/= (assignment operator) 27

:

: (vector generation operator) 5, 27

;

; (statement terminator) 3, 29

=

= (assignment operator) 27
== (equality operator) 6, 25

?

? (statement terminator) 3, 29

@

@ (element-by-element multiplication operator) 24
@= (assignment operator) 27

[

[ ] (element reference) 22

|

| (or operator) 6, 26
|| (or operator) 6, 27

"

" (quote character) 32

+

+ (addition operator) 25
+ (affirmation operator) 24
+= (assignment operator) 27

>

> (greater than operator) 6, 25
>= (greater or equal operator) 6, 25

^

^ (power operator) 23

\

\ (backslash character) 32

<

< (less than operator) 6, 25
<= (less or equal operator) 6, 25

A

Absolute file name 77
Absolute value 35
Addition operator 25
Affirmation operator 24
algae.out 89, 94
`ALGAE_SRC_PATH` environment variable 77
And operator 6, 26
Apostrophe 23
Append operator 4, 27
Arc cosine 35
Arc cosine hyperbolic 35
Arc sine 35
Arc sine hyperbolic 35
Arc tangent 36
Arc tangent hyperbolic 36
Argument 35
Arguments of functions 9, 22, 29
Arguments, command line 93
Arnoldi method 60
ARPACK 60
Arrays 32
Assignment operators 27
Assignments 28
Associativity of operators 21
Automatic class conversion 3, 21
Automatic type conversion 21
Autosrc functions 73

B

Back substitution 54
Backslash character 32
Bandwidth 40
Bandwidth reduction 44
BCSLIB-EXT 54, 57
Beep 20
Beginning execution 93
Binary input 67
Binary output 68
Block diagonal 40
Block transpose 41
Boeing Computer Services 54, 57
Braces 5
Brackets 22
break statement 8
Brennan, Mike 1
Bugs 99
Builtin functions 9

C

Call by value 9
Calling functions 22
Case 91
Case of variable names 17
catch 9
Catenation, string 25
Ceiling 36
Change directory 74
Character constants 32
Character strings 90, 91
Character type 4, 32
Cholesky factorization 54
Circular shift 41
Circumflex 23
Class conversions 3, 21
Class of entities 3, 17, 31
Closing a file 67
Colon 5, 27
Comma 4, 27
Command line arguments 93
Comments 29
Complement 53
Complex conjugate 36
Complex type 4, 32
Concatenation, string 25
Condition number 56
Conjugate transpose operator 23

Conjugate, complex . . . 36
Constants . . . 32
continue statement . . . 8
Control-flow statements . . . 5
Conversions between classes . . . 21
Conversions between types . . . 21
Converting between classes . . . 3
Cosine . . . 36
Creating new members . . . 3
Cubic splines . . . 66

D

Data types . . . 32
Debug level . . . 94
Defining functions . . . 10, 29
Defining variables . . . 17
Deleting entities . . . 4
Dense storage . . . 42
Detours . . . 99
Diagonal of a matrix . . . 42
Digital filter . . . 59
Digits to print . . . 20, 67
Digits, significant . . . 29
Directory, changing . . . 74
Disassembler . . . 93
Division operator . . . 25
Dot operator . . . 3, 17, 23
Double quotes . . . 32
Dynamic linking . . . 73

E

Efficiency of loops . . . 7
Efficient matrix products . . . 63
Eigenvalues and eigenvectors . . . 55, 60
Element numbering in matrices . . . 5
Element numbering in vectors . . . 4
Element references . . . 22
Emacs . . . 95
End of file . . . 14
Entities . . . 3, 17, 31
Entity members . . . 3, 17
Environment variables . . . 75
Equality operators . . . 6
Equilibration . . . 56
Error function . . . 37
Error messages . . . 69, 95
Escape sequences . . . 32
Evaluation . . . 74, 75
Evaluation of variables . . . 18
exceptions . . . 8
Exceptions . . . 75
Exclamation mark . . . 24, 68
Execution profiling . . . 89, 94
Exiting . . . 75
Exponential . . . 37
Expressions . . . 17, 28

F

Factors, prime . . . 38
False . . . 6
Features . . . 75
FFTW . . . 58
File closing . . . 67
File searching . . . 77
Files on the command line . . . 93
Floor . . . 37
for statement . . . 7
Format for printing . . . 67
Format, real or complex number . . . 29
Formatted output . . . 68, 69
Forward substitution . . . 54
Fourier transform . . . 58
Full storage . . . 44
Function calls . . . 22
Function expressions . . . 29
Functions . . . 9, 29, 32
Future of Algae . . . 97

G

Generate operator . . . 5, 27
Generating vectors . . . 5
Gibbs-King . . . 44
Gibbs-Poole-Stockmeyer . . . 44
Global scope . . . 10
Global symbol table . . . 18
Global variables . . . 17
Greatest common divisor . . . 37
grep . . . 45

H

Hunziker, Jennifer ... 1
Hunziker, Scott ... 1
Hyperbolic arc cosine ... 35
Hyperbolic arc sine ... 35
Hyperbolic arc tangent ... 36
Hyperbolic cosine ... 37
Hyperbolic sine ... 39
Hyperbolic tangent ... 39

I

Identity matrix ... 45
if statement ... 5
Imaginary value ... 38
in keyword ... 7
Initial value problems ... 66
Input filters ... 68, 77, 95
Input, binary ... 67
Input, character ... 69
Input, numeric ... 69
Integer power program ... 14
Integer type ... 4, 32
Integration ... 66
Interactive mode ... 3, 94
Interpolation ... 66
Interpreter ... 93
Interrupt signal ... 96
Interrupting loops ... 8
Intersection ... 53
Inverse cosine ... 35
Inverse fourier transform ... 59
Inverse hyperbolic cosine ... 35
Inverse hyperbolic sine ... 35
Inverse hyperbolic tangent ... 36
Inverse tangent ... 36
Inverse, matrix ... 59

K

Killing Algae ... 96
Known bugs ... 99

L

Labels in submatrix expressions ... 5
Labels in subvector expressions ... 4
Lanczos method ... 60
LAPACK ... 54
Least common multiple ... 38
Least squares ... 63
Length of variable names ... 18
Lewis, John ... 44
Linear equations ... 54, 63
Linear spacing ... 46
Linking, dynamic ... 73
local statement ... 10, 18
Logarithm ... 38
Logarithmic spacing ... 47
Logical operators ... 6
Loops ... 7
Lower case ... 91
Lower triangle ... 52

M

Magic squares ... 47
Magnitude ... 35
Matching labels ... 4
MATLAB ... 1, 33, 71, 72
Matrices ... 5, 31
Matrix constants ... 33
Matrix factorization ... 54, 57
Matrix inverse ... 59
Matrix labels ... 5
Matrix multiplication ... 63
Matrix norm ... 49
Matrix products ... 63
Matrix, diagonal ... 42
Maximum ... 45, 47
Member names ... 23
Member references ... 23
Members ... 3
Members of entities ... 17
Members of matrices ... 31
Members of scalars ... 31
Members of vectors ... 31
Messages, error ... 95
Minimum ... 46, 48
Minus sign ... 24
Modulus operator ... 25
Monte Carlo integration ... 65
Multiplication operators ... 24

N

Names of members . . . 23
Names of variables . . . 17
Negation operator . . . 24
Newline . . . 3, 29
Nonlinear equations . . . 65
Norm . . . 49
Not operator . . . 6, 24
NULL . . . 4
Number reading program . . . 14
Numeric constants . . . 32
Numerical integration . . . 66

O

Operations on functions . . . 9
Operations on NULL . . . 4
Operations on tables . . . 5
Operators . . . 20
Operators, associativity . . . 21
Operators, precedence . . . 21
Options on the command line . . . 93
Or operator . . . 6, 26, 27
Ordinary differential equations . . . 66
Output filters . . . 68, 77, 95
Output, binary . . . 68
Output, formatted . . . 68, 69
Overdetermined systems . . . 63

P

Parentheses . . . 22
Passing arguments by value . . . 9
Path lists . . . 75
Phase . . . 35
PID . . . 20
Pipes . . . 68, 77, 95
Plus sign . . . 24
Polynomial roots . . . 66
Power operator . . . 23
Precedence of operators . . . 21
Predefined members . . . 3
Predefined variables . . . 20
Prime factors . . . 38
Prime numbers . . . 38
Print format . . . 67
Printing formatted output . . . 68, 69
Profile . . . 40
Profile reduction . . . 44
Profiling . . . 89, 94
Program, integer powers . . . 14
Program, number reading . . . 14
Program, temperature conversions . . . 13
Projects . . . 97
Prompt . . . 20, 95

Q

Question mark . . . 3, 29
Quit signal . . . 96
Quote character . . . 32

R

raise an exception . . . 8
Random numbers . . . 49, 50, 51
Reading entities . . . 71
Reading numbers . . . 69
Reading text . . . 69
Readline facility . . . 95
Real type . . . 4, 32
Real value . . . 39
Recursion . . . 10
Reduction, bandwidth . . . 44
Reduction, profile . . . 44
Referencing elements . . . 22
Referencing members . . . 23
Regular expressions . . . 45
Relational operators . . . 6, 25
Relative file name . . . 77
Reported bugs . . . 99
Reporting bugs . . . 99
Restricted execution . . . 95
return statement . . . 10
Root finding . . . 65
Roots of a polynomial . . . 66
Rounding . . . 39
Runge-Kutta . . . 66

S

Scalar constants . . . 32
Scalars . . . 31
Scope of variables . . . 10, 18
Searching . . . 45
self . . . 10
Semicolon . . . 3, 29
Separating arguments . . . 9
Sets . . . 53
Shared objects . . . 73
Shift, circular . . . 41
Short And . . . 26
Short Or . . . 27
SIGINT . . . 96
Sign . . . 50
Signals . . . 96
Significant digits . . . 29
Signum . . . 50
SIGQUIT . . . 96
Sine . . . 39
Singular value decomposition . . . 64
Solving linear equations . . . 54, 63
Solving nonlinear equations . . . 65
Sorting vectors . . . 46, 50
Source files . . . 77
Sparse storage . . . 51
Spline interpolation . . . 66
Splitting into tokens . . . 90
Square root . . . 39
Standard functions . . . 35
Standard input . . . 94
Starting algae . . . 3
Startup files . . . 93
Statement terminators . . . 3, 29
Statements . . . 3, 17, 29
Statements for control-flow . . . 5
stderr . . . 69
stdin . . . 94, 95
String concatenation . . . 25
Strings . . . 4, 32, 90
Submatrix expressions . . . 5
Subspace iteration . . . 64
Substrings . . . 91
Subtraction operator . . . 25
Subvector assignment . . . 4
Subvector expressions . . . 4
SuperLU . . . 54
Symbol table . . . 18

T

Tables . . . 5, 31
Tangent . . . 39
Temperature conversion program . . . 13
Temporary files . . . 70
Terminal, width . . . 20
Terminating Algae . . . 75
Terminating statements . . . 3, 29
Testing for equality . . . 6
Tokens, splitting . . . 90
Transpose operator . . . 23
Trapezoidal integration . . . 66
Triangle . . . 52
True . . . 6
try . . . 8
Type conversions . . . 21
Types . . . 4, 32

U

Unary minus operator . . . 24
Unary plus operator . . . 24
Underdetermined systems . . . 63
Union of two sets . . . 53
Upper case . . . 91
Upper triangle . . . 52
User functions . . . 9

V

Variable evaluation . . . 18
Variable names . . . 17
Variables . . . 3, 17
Variables, environment . . . 75
Variables, predefined . . . 20
Variables, scope . . . 18
Vector constants . . . 33
Vector generation . . . 5, 27
Vector labels . . . 4
Vector norm . . . 49
Vectors . . . 4, 31
veil statement . . . 19
Version . . . 95
vi . . . 95

W

while statement . . . 7
Width of terminal . . . 20
Wrapping lines . . . 20
Writing entities . . . 72

Function Index

A

abs 35
acos 35
acosh 35
all 88
arg 35
asin 35
asinh 36
atan 36
atan2 36
atanh 36
atof 88
autosrc 73

B

backsub 54
band 40
bdiag 40
brent 65
btrans 41
builtin 73

C

cd 74
ceil 36
char 88
chol 54
circshift 41
class 89
close 67
combine 42
complement 53
conj 36
cos 37
cosh 37
cram 42

D

dense 42
diag 42
dice 42
diff 43
digits 67

E

eig 55
equal 89
equilibrate 56
erf 37
erfc 37
eval 74
exception 75
exec 75
exit 75
exp 37
exsparse 43

F

factor 57
fft 58
file 75
fill 43
filter 59
find 43
first 44
floor 37
form 44
fprintf 68
fread 67
full 44
fwrite 69

G

gcd 38
get 71
get_path 75
getdyn 71
getenv 75
getmat 71
gpskca 44
grep 45

H

hermitian 45

I

ident 45
ifft 59
imag 38
imax 45
imin 46
info 89

integer 38
intersection 53
inv 59
iram 60
isort 46

L

label 46
last 46
lcm 38
leastsq 63
line 75
linspace 46
load 72
log 38
log10 38
logspace 47
lose 47

M

magic 47
matrix 47
max 48
members 89
merge 48
message 69
min 48
mksparse 48
monte 65
mult 63

N

norm 49
npsol 78

O

ode4 66

P

pick 49
plot 85
primef 38
primes 39
print 69
printf 69
product 49
prof 89
provide 75
put 72
putdyn 72
putmat 72

R

rand 49
randn 50
read 69
readnum 70
real 39
replot 86
require 76
reverse 50
roots 66
round 39

S

save 73
scalar 50
search_path 76
select 50
seq 50
set 53
shape 50
show 90
sign 50
sin 39
sinh 39
solve 63
sort 50
source 76
sources 76
sparse 51
spline 66
split 90
splot 87
sprintf 70
sqrt 39
srand 51
src 77
ssi 64
string 90
strip 77
substr 91
sum 51
surprise 51
svd 64
symmetric 52
system 78

T

tan 39
tanh 39
test 91
time 91
tmp_file 71
tolower 91
toupper 91
transform 64
trapz 66
tril 52
triu 52

U

umin 87
union 53
unlabel 52
unplot 88

V

vector 52

W

what 92
who 92

Z

zero 52

Table of Contents

1 About Algae **1**

2 Primer **3**

3 Example Programs **13**

3.1 Temperature Conversion 13
3.2 Reading and Summing Numbers 14
3.3 Integer Powers 14

4 The Algae Language **17**

4.1 Entities 17
4.2 Variables 17
4.2.1 Variable Names 17
4.2.2 Evaluation of Variables 18
4.2.3 Scope of Variables 18
4.2.4 Predefined Variables 20
4.3 Operators 20
4.3.1 Function Calls 22
4.3.2 Element References 22
4.3.3 Member References 23
4.3.4 Transpose 23
4.3.5 Power 23
4.3.6 Not 24
4.3.7 Negation 24
4.3.8 Multiplication 24
4.3.9 Addition 25
4.3.10 Relation 25
4.3.11 And 26
4.3.12 Or 26
4.3.13 Short And 26
4.3.14 Short Or 27
4.3.15 Generate 27
4.3.16 Append 27
4.3.17 Assign 27
4.4 Expressions 28
4.5 Statements 29
4.6 Functions 29

5 Data **31**

6 The Standard Functions **35**

6.1 Basic Math 35
6.2 Arrays 40
6.3 Sets 53
6.4 Linear Algebra 54
6.5 Numerical Analysis 65
6.6 Basic I/O 67
6.7 Entity I/O 71
6.8 Execution 73
6.9 Special Tools 78
6.10 Miscellaneous 88

7 Running Algae **93**

7.1 Startup Files 93
7.2 The Command Line 93
7.3 Errors 95
7.4 The algae Prompt 95
7.5 Signals 96

8 Projects **97**

9 Bugs **99**

9.1 Reporting Bugs 99
9.2 Reported Bugs 99

Concept Index **101**

Function Index **107**