An Interface From S to Mathematica

Javier F. Cabrera

Rutgers University

Allan R. Wilks

AT&T Bell Laboratories

ABSTRACT

This paper describes a new interface in the S language to Mathematica. The interface allows the S user to have direct access to Mathematica through a co-process mechanism. It effectively extends the capabilites of S in two areas; high precision numerical computations and symbolic computations. A number of examples of the interface are given, including differentiation, integration and maximum likelihood estimation.

January 25, 1993

An Interface From S to Mathematica

Javier F. Cabrera

Rutgers University

Allan R. Wilks

AT&T Bell Laboratories

1. Introduction

S is a programming language whose original design was intended to facilitate interactive data analysis and graphics (Tukey, 1977). Today it has grown into a general purpose, interpreted, functional language that, like Mathematica, borrows strength from the design of many other languages. It is described in the book Becker, et al (1988), written by the authors of the language and has several tens of thousands of users worldwide, in industry, government, and academia. It is typically used in either a UNIX or DOS environment.

Though S and Mathematica are similar in a number of ways, their relative strengths are such that neither majorizes the other. For example, S is weak on symbolic computation relative to Mathematica, but its data structures are more tuned to typical numeric computation than those of Mathematica. In particular, the atomic objects of S are *vectors* (of numbers, character strings, ...) rather than single numbers (or character strings, ...). This has far-reaching implications on both the way users think of their computations, as well as the efficiency of them.

One of the main design principles of S is that of supplying *interfaces* to other systems. S, as distributed, has interfaces to C, Fortran, and UNIX. The reason for this design is the belief that it is better to leave to those other systems what they do well, rather than extend S to duplicate their capabilities. Two areas in which Mathematica's abilities exceed those of S are high precision numerical computations and symbolic computations. For these reasons, we have designed a new interface in S, to Mathematica. The interface is intended to utilize the numeric and symbolic capabilities of Mathematica, but does not attempt to be a complete translator between the two languages.

Some examples of symbolic computations utilizing the new interface are differentiation and integration of S expressions, as in

```
> e <- D(exp(-1/2*xˆ2), "x")
> e
-1 * ((exp((-1/2) * (xˆ2))) * x)
> Integral(e, "x")
exp((-1/2) * (xˆ2))
```

where the functions `D()` and `Integral()`, given later, utilize the interface to perform their computations. Users of Mathematica should have little trouble reading S expressions by keeping in mind a few simple mental translations. For example, S uses parentheses for function application (yes, and for grouping!) and a two-symbol, left-pointing arrow for assignment. The `>` symbol is the S prompt, so the remainder of lines beginnning with it are user input.

The following example illustrates a (somewhat trivial) high-precision computation. First the value of π is computed to 30 places using an appropriate Mathematica expression. This is a simple illustration of the low-level interface, `Math()`, to Mathematica. The return value of `Math()` is always a character vector of length 1. Each object in S is a vector and has a mode (this is analogous to lists and heads in Mathematica). If the mode is `character` then the elements of the vector are character strings (not characters!) so that a character vector of length 1 is a vector consisting of a single string. There is a general type-extension mechanism in S based loosely on the notion of object-oriented *classes* and using this we have designed a new class, `bigfloat`, that encapsulates the idea of high-precision numbers. The return value from `Math()` below is coerced into an object of class bigfloat and then printed out. Associated with each class in S are a number of *methods*, which are implemented as functions whose actual name ends with a dot, followed by the class name, but which can be invoked by naming the function without that suffix (the *generic* function). There are methods for arithmetic with `bigfloat` numbers, and this is shown in the next expression, where we compute the reciprocal of 2π. Finally, we show the (double precision) value of $1/2\pi$ known to S, with a few extra garbage digits thrown in for good measure.

```
> Pi30 <- as.bigfloat(Math("N[Pi,30]"))
> Pi30
[1] 3.14159265358979323846264338328
> 1/(2 * Pi30)
[1] 0.159154943091895335768883763337
> print(1/(2*pi), digits=20)
[1] 0.15915494309189534000
```

Notice here the use of = in S for naming arguments in function calls. Later we will see examples of = used in function definitions; this indicates an optional argument, and the expression to the right of it is the default value for that argument. A word about the `[1]` that appears at the beginning of the output lines: when a vector is printed in S its elements are printed several to a line, and each line begins with the (1-based) index, in square brackets, of the first element of that line.

In the following sections we describe the various components of the interface, beginning in the next section with the lowest level—communication between S and Mathematica. In sections 3 and 4 we show how we interface to the numeric and symbolic parts of Mathematica. The final section gives further examples.

2. Low-Level Interface

Communication between S and Mathematica is done at the level of a stream of characters: a Mathematica expression is composed in S, sent for evaluation to Mathematica and the (normally printed) result is returned to S as a character vector (of length 1). This can be done in several ways. The simplest is to use the S function `unix()` each time a Mathematica expression is to be evaluated:

```
Math <- function(expr) unix(paste("math", expr))
```

The `paste` function in S coerces each of its arguments to a character vector and pastes these vectors together. Think of writing the vectors of character strings in columns (there is a general rule for when the columns are of unequal length, but we won't need it here) and then just pasting the rows together, separated by single spaces (this is settable by an optional argument `sep`). In the example just given, each argument to a`paste`is simply paste the two character strings together, separated by a space. The resulting string is sent to a UNIX shell.

While simple, this approach suffers from the disadvantage that Mathematica must be invoked each time `Math()` is called. A better, though more complicated, approach is to invoke Mathematica once, as a co-process, and effect all communication between S and Mathematica via a pair of UNIX pipes. This is the approach we have taken; the pipes are established by a C routine `math()` that gets loaded (statically or dynamically) with S. The action of `math()` is simple: if Mathematica is not running, `math()` starts it. In any case, a single line of input is sent to Mathematica, and a single line of output is read in return. The input can be several Mathematica expressions, but they must be combined into one line, and they must cause only one line of output. When `math()` starts Mathematica, it does so in such a way that output expressions won't be broken into several lines.

The calling sequence for `math()` is:

```
math(start_command, command, output, maxbuf, force_start, error)
char *start_command, *command, *output;
long maxbuf, force_start, error;
```

Here `start_command` is a character string giving the command to use if Mathematica is to be started, `command` is the command to be sent to Mathematica, `output` is a character string of length `maxbuf` used to stuff the return value from Mathematica, `force_start` is a flag, which, if true, indicates that Mathematica should be (killed first if it's already running, and) started, and `error` is an attempt to propagate back to S an error found by Mathematica.

In general, errors are quite difficult to handle, primarily because they don't appear in a fixed way. Even though the Mathematica co-process is running in a clean way—one line in, one line out—error messages can have a variety of formats, from one line to many lines. For example, some syntax errors cause a line to be printed with the string `"^ <retype line>"` beginning under the offending spot in the input command, followed by a line with some spaces at the beginning and no newline. Currently, `math()` detects these sort of errors, setting `error` to –1 on return (normal return is 0).

The other major type of errors are those that occur while Mathematica is evaluating an expression. Empirically, the printing of such an error message is preceded by a blank line, which won't normally happen in the way we have Mathematica running. Hence, if a blank line is detected, `math()` will attempt to read a following error message, and will return an `error` value of –2. This process is not guaranteed to be successful—since error messages can be a variable number of lines, it is impossible to know when Mathematica has finished printing its message. Currently, `math()` tries to read the lines of the error message, pausing for a second between each line to allow it to appear on the pipe. If not all lines get read, then unread lines will appear to be the answer to the next call to `math()`. When this happens, the recommended action is to restart Mathematica forcibly.

There are other sorts of errors as well. For example, many typos do not cause either a syntax or a run-time error since mistyped names are taken by Mathematica as variables or undefined functions. In those cases Mathematica returns the unevaluated expression, which in turn is returned by `math()`. The new MathLink protocol may relieve somewhat the burden of the low-level interface, though it is unclear at this time whether MathLink will make error handling any easier.

The C routine `math()` is called by the S function `Math()`:

```
Math <- function(command = "", force.start = F,
     startup = "math -noprompt -run Get@ToString@SMathStart") { ... }
```

The optional argument `force.start` (note the use of `=` to establish a default value in the function definition) allows the user to restart Mathematica forcibly—this could be necessary, for example, if `math()` did not successfully read a complete run-time error message. The optional argument `startup` is a UNIX command that will be used to initiate the Mathematica co-process; the default value turns off Mathematica's prompts and runs a startup script found in the file `SMathStart`. The form of this command, with the `@` characters, is chosen to avoid using double quotes. Here, for reference, is the `SMathStart` file:

```
Format[StringBreak[n_]] := "";
Format[Continuation[n_]] := "";
Format[LineBreak[n_]] := "";
ResetMedium[PageWidth->Infinity];
$PrePrint = InputForm;
<<S.m
```

The file `S.m` contains a set of Mathematica definitions that will be discussed in section 4.

Typically, `Math()` will be called from other functions, as we shall see. But it can be called explicitly, as illustrated in the Introduction and here:

```
> Math(1)
[1] "1"
> Math("N[Pi,30]")
[1] "3.14159265358979323846264338328"
> Math("D[Sin[x],x]")
[1] "Cos[x]"
```

3. Computations With Numbers

The first area in which Mathematica can be used to extend the capabilities of S is in high-precision calculations. S uses standard machine data types, usually an integer type and two sizes of floating point types. Mathematica adds to this the ability to deal with arbitrary integers (subject to memory constraints, of course) and high-precision floating point numbers. In S, we have created a new class (see the Appendix of Chambers (1991)) of `bigfloat` objects. A `bigfloat` vector is a representation of high-precision floating point numbers. The actual representation is as a `character` vector giving the ASCII representation of the decimal form of the individual numbers.

There are several advantages of this representation. Printing such numbers is easy—the printing method, `print.bigfloat`, just calls the standard printing function for character vectors, but with the automatic printing of double quotes around each string turned off. The S code that does this is

```
print.bigfloat <-
function(x)
{
        class(x) <- NULL
        print.default(x, quote=F)
}
```

This just uses the default printing method on `x`, after removing its class to prevent recursive calling of the `bigfloat` printing method. Another small advantage of using ASCII representations of numbers is that they are then (almost) already in a form suitable to be sent to Mathematica. `bigfloat` vectors can be initially created by using the `as.bigfloat` function mentioned in the Introduction, which typically takes as input a numeric vector and converts it to the internal form for a `bigfloat` vector.

In order to develop more significant methods, such as those for arithmetic and trig functions, it helps to establish a standard form for the way in which Mathematica functions will be called. Mathematica (like S) employs a variety of syntaxes, but has the property that all expressions can be written in a standard functional form. In addition Mathematica (again, like S) has a lambda-calculus-like ability to create function definitions on the fly. We will exploit these properties by mapping S functions into Mathematica functions, some of which will have temporary definitions. To this end we create a character vector in S whose elements are Mathematica functions and whose `names` attribute gives the names of the S functions that are attempting to be mimicked by the Mathematica functions. (Any S object can have *attributes*. For example, if a vector has a `dim` attribute, it should be an integer vector the product of whose elements is the length of the vector. In this case, S treats the vector as an array with dimensions as given by the attribute. Any vector can have a `names` attribute, which should be a character vector the same length as the original vector, and giving names to each of its elements.) Here is the beginning of that vector:

```
Math.functions <-
c(
        abs="Abs", acos="ArcCos", acosh="ArcCosh",
        asin="ArcSin", asinh="ArcSinh", atan="ArcTan",
     ...
        trunc="Times[Sign[#],Floor[Abs[#]]]&",
     ...
)
```

(The names are specified before the `=` signs, and the `c()` function in S merely combines its arguments into a single vector, similar to the use of braces to collect lists in Mathematica.) Notice that many S functions have direct analogs in Mathematica. For each element of `Math.functions`, it is expected that a correct Mathematica expression will result by following the function string with a left square bracket, one or more comma-separated arguments, and a right square bracket. So, for example, the Mathematica version of the S function `trunc()` (for truncating numbers towards 0) constructs a string that can stand as a function of one argument. Our current version of `Math.functions` has about 60 translations between S functions and Mathematica functions.

We are almost ready to show some methods for `bigfloat` vectors, but it will help to have a few utilities that construct simple Mathematica objects. For example, we often want to put the elements of a vector into a single string, separated by commas:

```
comma.sep <- function(x) paste(x, collapse=",")
```

The options `collapse` argument to `paste` collapses the pasted strings into one composite string, using the value of the argument as the ‘‘glue’’ between the pieces. In the example above, there is actually no pasting to do, as there is only one unnamed argument, so the effect is just to concatenate the elements of `x` with commas between them. In conjunction with this is a function to make up a Mathematica list:

```
Math.list <- function(x) paste("{", comma.sep(x), "}")
```

Mathematica numbers will be read and written in scientific notation like:

```
9.87983344*10^6
```

and we will want to change back and forth between this format and the format used by S:

```
9.87983344e6
```

In addition, we will supply vectors of numbers to Mathematica (and get them back) in the form of Mathematica lists. We have two functions, `Math.numbers()` and `S.numbers()` that do the translations, and they are calls into some C code. We originally implemented these as calls to the UNIX `sed` command through the `unix()` function, but because they are used so much, this slowed down the interface significantly. Here are examples of their use:

```
> Math.numbers(exp(50:60))
 [1] "5.184705528587072*10^+21"  "1.4093490824269391*10^+22"
 [3] "3.8310080007165788*10^+22" "1.0413759433029096*10^+23"
 [5] "2.8307533032746935*10^+23" "7.6947852651420184*10^+23"
 [7] "2.0916594960129971*10^+24" "5.6857199993359364*10^+24"
 [9] "1.5455389355901036*10^+25" "4.2012104037905148*10^+25"
[11] "1.1420073898156847*10^+26"
> S.numbers(Math("Exp[Table[n,{n,50.0,60.0}]]"), 11)
 [1] 5.184705528587072e21 1.409349082426939e22 3.831008000716577e22
 [4] 1.041375943302909e23 2.830753303274694e23 7.694785265142017e23
 [7] 2.091659496012996e24 5.685719999335932e24 1.545538935590104e25
[10] 4.201210403790515e25 1.142007389815684e26
```

Notice, by the way, the use of the `:` operator in the first expression to generate a sequence of numbers. One final utility that will be useful later is `Math.names`, which is again written in C. This maps a legal S name into a legal Mathematica name. Since these are almost the same there is little to do; in fact, the only thing it does is to change `.` into `$`.

We are now ready for a more complicated method. As mentioned earlier, if `x` has class `bar` then the function call `foo(x)` causes the method for `foo` to be invoked on `x` and this is implemented in the function `foo.bar`. This would make for a great proliferation of methods, if we had to write a separate function for each arithmetic operator, trig function, etc. To avoid this, S collects a number of these standard functions into three groups, called the ‘‘Ops’’, ‘‘Math’’, and ‘‘Summary’’ methods. A method written for one of these (`Ops.bar`, for example) will be invoked for a number of S functions. The first of these is used for infix operators, the second for special functions (like exponential and trig functions) and the third for a collection of summary functions, such as `sum`, which sums the elements of its argument. (The name collision of the `Math()` function, given earlier, and the ‘‘Math’’ method, is unfortunate.) To see how this works, we give here a simplified version of `Math.bigfloat` that doesn’t try to deal with the 3 of the 22 ‘‘Math’’ functions that take two arguments:

```
Math.bigfloat <-
function()
{
     mathfun <- Math.functions[.Generic]
     matharg <- Math.list(Math.numbers(x))
     command <- paste(mathfun, '[', matharg, ']')
     as.bigfloat(S.numbers(Math(command), length(x)))
}
```

The name of the function, given as `.Generic`, is used to look up the corresponding Mathematica function name in the `Math.functions` vector we constructed earlier. The rest of the function is straightforward, using the utilities described previously. The full version deals with the 2-argument ‘‘Math’’ functions and also makes sure that a matrix input results in a matrix output.

Simple arithmetic can also be done with `bigfloat` vectors, by using the `Ops.bigfloat` method:

```
Ops.bigfloat <-
function()
{
        mathfun <- Math.functions[.Generic]
        n <- max(length(e1), length(e2))
        matharg1 <- Math.list(Math.numbers(e1)))
        matharg2 <- Math.list(Math.numbers(e2)))
        command <- paste(mathfun, '[', matharg1, ',', matharg2, ']')
        as.bigfloat(S.numbers(Math(command), n))
}
```

This method is very similar to the previous one, differing only in the fact that it deals with two arguments rather than one. Again, the actual method is a little more complicated, as it must deal with the possibility of array inputs, and inputs of different lengths.

The method for the ‘‘Summary’’ functions, like `sum()` and `prod()` looks exactly like that given above for the ‘‘Math’’ functions. Additional methods can be written for individual functions, as desired; with the tools given above this is usually easy.

What is the precision of a `bigfloat` number? An entire `bigfloat` vector could be assigned a precision, or individual members could. In either case the precision could be carried explicitly as an attribute of the vector, or implicitly by the number of digits present in the representation.

We have chosen the latter approach because there seems to be little reason to manipulate the precision in S, so it's just a burden to carry it around explicitly. Mathematica has a comprehensive set of rules regarding precision, and these rules will be invoked according to the number of digits in the numbers passed to it. This method of carrying precision must be kept in mind when using the interface. For example, to compute the 100th power of 2:

```
> as.bigfloat(2) ˆ 100
[1] 1.267650600228229e30
> as.bigfloat("2.00000000000000000000000000000") ˆ 100
[1] 1.2676506002282294014967032054e30
> as.bigfloat("2.000000000000000000000000000000000000") ˆ 100
[1] 1267650600228229401496703205376.000000
```

In the first expression, Mathematica saw a floating-point 2 and assumed it had double precision, giving a double precision answer. In the second and third cases, the number of digits in the answer matches the number of digits in the `bigfloat` created by `as.bigfloat()`. The `as.bigfloat()` function should probably have an optional argument, allowing the selection of the precision at the time of creation of the numbers.

The example above would be better accomplished by having a `bigint` class of numbers in S. It is certainly possible to create such a class, but we have not yet done so.

4. Computations With Expressions

In addition to numerical computations we want to take advantage of Mathematica's symbolic capabilities to perform computations with S expressions. There are a number of possible applications of this capability, as illustrated in the next section. Each of these seems to follow four steps in their implementation.

First we need to bring the state of Mathematica into alignment with S; in other words, if an S expression is to be sent to Mathematica, some of the variables in that expression may have data associated with them, and if so, we may wish to pass this data on to Mathematica. Second we must translate the S expression into a Mathematica expression before sending it through `Math()`. Third, we must translate the Mathematica answer back to an S expression. Finally, in order that the Mathematica state established in the first step doesn't carry over to subsequent expressions (the corresponding S datasets may have been removed), Mathematica must be told to remove the definitions of those variables.

The following subsections describe these steps in detail, using as an example a general S function, `math()`, that deals with Mathematica at the level of expressions, using `Math()` for it's communication.

4.1. Assignments

When Mathematica evaluates an expression, names without any preassigned value are treated as symbols while names assigned to data structures are replaced by their values. Thus, before an S expression is evaluated by Mathematica, we must assign to the names contained in the expression their corresponding values in S, if any.

In `math()` this task is performed with a call to two functions. The first one is `all.objects(expr)` which returns a list of all the names in the expression `expr` that have current values in S. For example, let's apply the function `all.objects()` to an expression containing names such as `y` which have not been assigned any value, and others such as `x` which have been assigned a value.

```
> x <- 2
> all.objects(expression(sin(pi*x) * cos(pi*y)))
[1] "pi" "x"
```

This function first calls the function `all.names()` which returns all the names in the expression. In this example it gives:

```
> all.names(expression(sin(pi*x) * cos(pi*y)))
[1] "*"   "sin" "*"   "pi"  "x"   "cos" "*"   "pi"  "y"
```

`all.objects()` then successively deletes from this list of names those that fail to satisfy certain predicates. In particular, it checks that the names correspond to existing objects that are not functions and that the names are not one of the logical constants or `NULL`, or in the list given in the optional second argument `except`. In the above example, the names `sin`, `cos`, and `*` where expunged from the list of names because they correspond to S functions, `y` was deleted because it didn't have a previous value, and `x` and `pi` where kept. If we were not interested in allowing `pi` to be returned by `all.objects()` we could use it as second argument as follows:

```
> all.objects(expression(sin(pi*x) * cos(pi*y)), except="pi")
[1] "x"
```

As we will see, this second argument is useful to guarantee that, for example, a variable with respect to which we are integrating does not get assigned a value in Mathematica just prior to the integration.

The second function involved in updating the state of Mathematica is `math.assign()`, which has Mathematica execute a set of assignments that leave the variables discovered by `all.objects()` with the same values in Mathematica as they have in S. For example the call

```
> math.assign("x")
```

will assign to `x` in Mathematica the same value that it already has in S. We can check this by the call

```
> Math("x")
[1] "2"
```

There is also an optional second argument to `math.assign()` which allows the value of the variables in the assignments to be set explicitly, rather than being picked up automatically (via `get()`) in the S environment.

The actual work of creating a Mathematica expression that evaluates to the value of a given S object is done by `Math.array()` which is an extension of `Math.list()` given earlier. Here is an example with a simple matrix.

```
> x <- matrix(1:12, 3, 4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> Math.array(x)
[1] "{ { 1,4,7,10 },{ 2,5,8,11 },{ 3,6,9,12 } }"
```

`Math.array()` uses `Math.list()` for non-array vectors and in the case of an array, it calls itself recursively, creating the Mathematica representation of the array as a nested set of lists.

The fourth step in the paradigm described in the introduction to this section is to back out the assignments made in Mathematica. This is easily accomplished with the function `math.remove()`:

```
math.remove <-
function(objnames)
     if(length(objnames) > 0)
          Math(paste("Remove[", comma.sep(objnames), "];"))
```

In the above examples, the assignment made to x can be removed with the call

```
> math.remove("x").
```

Again, this can be verified with the call

```
> Math("x")
[1] "x"
```

5. S to Mathematica

The function `as.math()` takes as argument an S expression and returns a string that is its translation to a Mathematica expression. The code of this function is displayed in Listing 1.

The function returns a value based on the mode of its expression argument. Most of the cases are self-explanatory; the most interesting one is the default case where the name of the function is used to index the `Math.functions` table, and the corresponding Mathematica function is used to create a Mathematica function call whose arguments are constructed as (recursive) calls to `as.math()` on the subexpressions of the input expressions. Let's walk through a simple example:

```
> as.math(expression(sin(x)))
[1] "Sin[x]"
```

Since the mode of `sin(x)` is `call`, (the mode in S for an expression that represents a function call) the corresponding Mathematica name from the `Math.functions` table is `Sin`. Then `as.math` is called again with `x` as argument. Since the mode of `x` is `name` then the recursion ends and `Sin[x]` is returned. Other more complicated examples are

```
> as.math(expression(exp(x^2)))
[1] "Exp[[Power[x,2]]"
> as.math(expression((y+sqrt(5))/2))
[1] "Times[#1,Power[#2,-1]]&[Plus[y,Sqrt[5]],2]"
> as.math(expression(Integrate(x^2,x)))
[1] "Integrate[Power[x,2],x]"
```

5.1. Mathematica to S

Once Mathematica evaluates the expression, the resulting output is translated to an S expression by the Mathematica function `SForm[]`, whose definition comprises the major portion of the file `S.m` mentioned in section 2.

Mathematica supports two functions, `FortranForm[]` , and `CForm[]`, which translate a Mathematica expression to its Fortran or C form. However since `SForm` is not meant to be a general purpose function but rather an internal function of the interface, its implementation differs from that of `FortranForm[]` , and `CForm[].`

For instance `SForm` tries to match Mathematica calls with S calls as much as possible, even if both calls are not equivalent in a more general context. However this is not true for `FortranForm[]` , and `CForm[]` which only match very simple calls such as the four arithmetic operations. For example, the value of `FortranForm[Sin[x]]` is `Sin(x)` which requires the users to define the function `Sin()` in their own fortran program. However the value of `SForm[Sqrt[x]]` is `sin(x)` which is a call to a function already defined in S. This particular call to `SForm` is implemented by the following definition.

```
SForm[Sin[ x_]] := StringJoin["sin(",SForm[x],")"]
```

This is one of multiple definitions in the Mathematica package S.m which make up the function `SForm[]`.

A second difference is in the translation of numbers. From the point of view of the interface there are two different types of numbers, double precision numbers and bigfloats. Therefore `SForm[]` calls the Mathematica function `Precision[]` and attempts to differentiate between machine numbers (double precision) and bigfloats and then makes the corresponding translation. This is implemented by the following definition of SForm

```
SForm[ x_Real] := If[Precision[x] <17,ToString[InputForm[x]],
     StringJoin["as.bigfloat(\"",ToString[InputForm[x]],"\")"]]
```

However for integer arguments Precision[] returns the value Infinity, so it cannot detect when an integer should be assigned class bigfloat. To overcome this we calculate the number of digits explicitly:

```
SForm[ x_Integer] := If[ Ceiling[N[Log[Abs[x]]/Log[10.]]] <17,
     ToString[InputForm[x]],
     StringJoin["as.bigfloat(\"",ToString[InputForm[x]],"\")"]]
```

Another important case is the translation of Mathematica lists. In Mathematica a numerical array is no different than a list, while S differentiates between lists of numbers and vectors or arrays of numbers. The function NumericArrayQ[] check out that all the elements of the list are numbers and that the list is a rectangular array.

```
NumericArrayQ[lst_List] := NumberQ[ Apply[Plus,Flatten[lst]] ] &&
                           TensorRank[lst]== Depth[lst]-1
```

If the list satisfies these two conditions it is translated to an S array, otherwise it is translated to an S list. The function ToNumericArray[] does the translation from a list to a numerical array. First it computes the dimensions of the array and then computes the maximum precision over the numbers in the array. If the maximum precision is greater than 17 it translate it to an S array of bigfloats, otherwise it returns the translation to double precision numbers. Here are two examples of this.

```
In[2]:= x

Out[2]= {{1, 2}, {3, 5}}

In[3]:= ToNumericArray[x]

Out[3]= array(c(1, 3, 2, 5),c(2, 2))

In[4]:= y

Out[4]= {{3.14159265358979323846264 3, 6.28318530717958647692528 7},

>    {9.42477796076937971538793, 15.70796326794896619231322}}

In[5]:= ToNumericArray[y]

Out[5]= as.bigfloat(array(c("3.141592653589793238462643","9.42477796076937971
>    538793","6.283185307179586476925287","15.70796326794896619231322"),c(2,
>    2)))
```

The important point here is that SForm[] utilizes the Mathematica parser to do the translation. We could try to do the reverse translation in S, by substituting squared brackets with parenthesis and let the S parser walk the Mathematica parse tree. But we would have have to add some C code for doing the conversion to bigfloats.

5.2. The Paradigm in Action

Now we are ready to put all the pieces together and show the general purpose function math():

```
math <-
function(expr)
{
      expr <- substitute(expr)
      expr.names <- all.objects(expr)
      math.assign(expr.names)
      value <- sform(as.math(expr))
      math.remove(expr.names)
      eval(value)
}

sform <-
function(expr.math)
      parse(text = Math(paste("SForm[", as.math(expr.math), "]")))[[1]]
```

The first line prevents evaluation of the input expression. We use `all.objects()` to find the objects to be assigned in Mathematica prior to the expression and `math.assign()` to construct and execute the assignments. Next, the auxiliary function `sform()` is used to translate the input expression into Mathematica using `as.math()` and then has Mathematica use `SForm[]` to translate the result back to S. The return value is a character string, which can be used as input to the S parser `parse()`, which in turn produces a vector of mode `expression` whose first element is the desired expression. A call to `math.remove()` backs out the assignments made by `math.assign()`

The final line of the function causes the returned expression to be evaluated. The action here depends on the application, as we shall see.

Some examples are:

```
> x <- 2
> math(sin(x))
> math(exp(-x*x/2)/sqrt(2*pi))
```

This translates and evaluates S expressions so it doesn't do anything new that S can't do. So we need to call on specific capabilities of Math that are not in S. We do this in the next section through a series of examples.

6. Examples

We now turn to several more detailed examples that follow the paradigm developed in the previous section. In each case the idea is the same: bring the state of Mathematica into line with that of S; perform the computation in Mathematica using `as.math()` in S and `SForm[]` in Mathematica to do expression translation; and finally, remove the state established in the first step.

6.1. Differentiation

The function `D()` (see Listing 2) is a simple modification of `math()`. It takes as arguments an S expression to be differentiated together with a character vector of variable names, and returns the partial derivative of the expression with respect to all the variables. Ignoring for the moment the second and third lines of the function body, we can see that the rest of the function is almost identical to that of `math()`. One difference is that the Mathematica expression is passed through `D[]` before being translated back to S. Notice also that the optional second argument to `all.objects()` is used to ensure that no prior assignment in Mathematica is made to any of the variables listed in the `vars` argument. The `if()` expression allows the first argument to be the name of a function call, rather than the call itself. For example:

```
> e <- D(x^3, "x")
> e
3 * (x^2)
> D(e, "x")
6 * x
```

Without the `if()` expression, the last result would have been 0, which is right if the first argument to `D()` is considered the constant function whose value is `e`.

6.2. Integration.

Another example in the Introduction was the function `Integral()` (see Listing 3) which has the same first two arguments as `D()` plus an optional third one giving limits of integration. There are three types of integrals that this function will compute. If no `lims` argument is given, it will compute the indefinite integral of `expr`. Otherwise it computes the definite integral. This can still be a symbolic expression, however, so the optional `numeric` argument says whether to force the value of the function to be a number. `Integral` is like `D()` except that it uses the Mathematica function `Integrate[]` or `NIntegrate[]` instead of `D[]`. There is some extra code for processing the `lims` argument, which should be a length-two vector for a simple integral, or a two column matrix with each row representing the limits of integration for one dimension of a multiple integral. The example given in the Introduction, was an indefinite integral. To demonstrate the definite integral let's compute the probability

$$\Pr\{1 < X_1 < 2,\ 1 < X_2 < 2\}$$

where $X = (X_1, X_2)$ has a bivariate normal distribution with $var(X_1) = var(X_2) = 1$, and $corr(X_1, X_2) = 0.3$. To obtain this probability we have to integrate the normal density function

$$\int_{\mathbf{R}^2} \frac{dx}{2\pi\sqrt{\det\Sigma}} \exp\{-\frac{1}{2} x^t \Sigma^{-1} x\} \quad ,$$

where Σ is the two by two matrix with diagonal elements equal to 1 and off-diagonal elements equal to 0.3.

```
> x <- c("x1","x2")
> sigma <- matrix(c(1.0, 0.3, 0.3, 1.0), 2, 2)
> dens <- substitute(exp(-x %*% sigma %*% x) / 2*pi*0.91)
> Integral(dens, x, matrix(c(1,1,2,2),2,2))
[1] 0.00121131
```

The `%*%` operator above is just matrix multiplication.

6.3. Maximum Likelihood Estimation

Much of statistical theory is concerned with the problem of estimation. In the simplest case it can be formulated as follows. Suppose we observe a sample $x_1 \ , \ldots , \ x_n$ generated by taking independent realizations of a random variable *X*, whose distribution F_θ depends on an unknown parameter θ. An estimator for θ is a function of the sample which, supposedly, is close to the value of the parameter θ.

One general method of constructing estimators is known as *maximum likelihood estimation* (MLE). The *log-likelihood* function is defined as the logarithm of the probability of the observed sample. For example, for independent observations,

$$L(\theta) = \log(\ f_\theta(x_1) \ \cdots \ f_\theta(x_N)\ ) = \sum_{i=1}^{N} \log f_\theta(x_i) \ ,$$

where f_θ is the density function of *X*.

The (MLE) is obtained by selecting the value of θ for which the likelihood (or log-likelihood) is greatest for the observed sample. This general method is important because it produces estimators which have desirable properties under mild regularity assumptions.

Cabrera (1989) showed a group of implementations of MLE problems using Mathematica. Listing 4 shows a simple S function that computes the MLE of a parameter. The arguments are: a density function, a set of variables, a set of parameters, a sample, and an initial parameter value for the iteration. The computation of the MLE of the parameter is done in a straightforward way, by minimizing the negative of the log-likelihood. For some problems one may have to do this computation more carefully by adding some simplifications, etc.

Since this example is a little more complicated, we have added some comments to the S code, showing that there are a couple of extra initializations in Mathematica, and that there are several steps to constructing the call to `FindMinumum[]` and then extracting the right piece of its result before return to S.

As an example, consider the Cauchy density function is given by the S expression

```
1/pi/(1+(x - theta)^2)
```

The likelihood function for `theta` does not always have a unique maximum, so we could give different starting points and observe the outcome:

```
data <- rcauchy(20)  # 20 random Cauchy variates
dens <- substitute(1/pi/(1+(x - theta)^2))
> MLEiid(dens, x, theta, data, 0)
[1] 0.0150443
> MLEiid(dens, x, theta, data, 0.5)
[1] 0.0151142
> MLEiid(dens, x, theta, data, -1.0)
[1] 0.0148650
```

One could now proceed by plotting the likelihood function in the region near 0.015 and investigate whether there is really more than one local maximum. Figure 1 shows a plot of the log-likelihood function with the vertical scale altered by a linear transformation. It clearly shows that the local maximum is unique. The dependence of the estimator of `theta` on the initial value is probably an indication of the flatness of the log-likelihood near the maxima. To verify this we change the parameter `AccuracyGoal` of the function `FindMinimum[]` from its default value of 6 to 10 by adding the argument `AccuracyGoal -> 10`. This results in changing the third-last line in the definition of `MLEiid()` to

```
value <- sform(paste("FindMinimum[", fun, ",", wrt, ",AccuracyGoal -> 10]",
                right.piece)) ,
```

which returns the value `0.0150433` for each of the the three initial values used earlier.

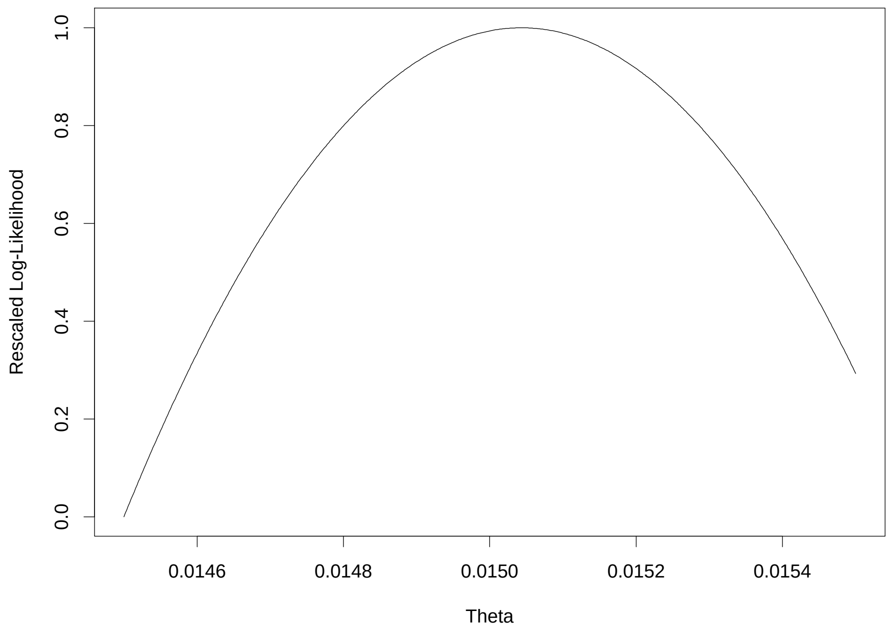

Figure 1. Log-Likelihood function for Theta near 0.015

6.4. Logistic Regression

A similar example to the last one is maximum likelihood estimation for the parameter of the logistic regression model. But here, instead of maximizing the likelihood function we will solve the likelihood equations, because it is a more efficient computation in Mathematica.

Suppose we observe a sample of pairs (y_i, x_i), $i = 1, \ldots, n$, where where x_i is a vector of covariates and y_i is zero or one. We are concerned with estimating the distribution of *y* given *x*. One possible model which relates the probability distribution of *y* to a linear function of *x* is the logistic model, defined by the formula:

$$logit(\text{Prob}(y=1)) \;=\; \beta^t x \;,$$

where the logit transformation is $logit(t) = t/(1-t)$. For a detailed description of the theory and methodology of logistic regression see McCullagh and Nelder(1983).

The log-likelihood function of the parameter vector β is proportional to

$$l(\beta) \;=\; \sum_{i=1}^{n} -y_i \;\beta^t x_i \;+\; \log(1 \;+\; \beta^t x_i) \;,$$

and the equivalent S expression is:

```
lik <- -y%*%(x%*%beta) + sum(log(1 + exp(x%*%beta)))
```

where `y` is the vector of y_i's , `x` is the matrix of rows x_i with a one added at the beginning, and `beta` is the vector of parameters.

The first line of the function `logist()` (Listing 5) adds a column of ones to `x`, the second one creates the vector of parameters `beta` and the third and fourth lines send the definitions of `x`, `y`, and `beta` to Mathematica. The log-likelihood is translated into a Mathematica expression by a call to `as.math()`. The Mathematica command which solves the likelihood equations is generated by the three assignments:

```
eqns   <- Math.list( paste("D[lik,",beta,"]==0") )
params <- comma.sep( paste("{", beta,",0}") )
value  <- paste("FindRoot[",eqns,",",params,"]",sep="")
```

which are passed to Mathematica for evaluation. Finally the datasets created in Mathematica are removed and the parameter estimate is returned. The following example demonstrates the use of `logist()`:

```
> x <- matrix(rnorm(30), 10, 3)
> y <- sample(c(0,1), 10, T)
> logist(x,y)
[1] -1.08798100 -0.08005744 -0.45785370 -1.67958400
```

The call `rnorm(20)` generates a sample of 20 normal random numbers (mean 0, variance 1), while the call to `sample()` samples 20 times (with replacement) from the vector `c(0,1)`, i.e., it computes a vector of 20 random bits.

6.5. The Bernoulli Shift

As a final example we now turn to one case that does not follow the paradigm developed in section 4, but rather it involves a calculation which requires high precision arithmetic.

The Bernoulli shift is the map

$$f(x) \;=\; 2x \;\text{mod}\; 1$$

from the interval [0,1] into itself. It has the property that $f(x)$ can be obtained from the binary representation of x by deleting the first digit after the decimal point and then shifting the remaining digits one position to the left. The sequence $x_n \;=\; f^n(x_0)$ is a well known example of a deterministic sequence which appears stochastic.

The S function

```
bernoulli <-
function(x, n)
{
     for(i in 2:n) x[i] <- (x[i - 1] * 2) %% 1
     x
}
```

returns the sequence of length `n` with initial value at `x`.

If `x` is a double precision number the sequence becomes zero after 55 numbers since all the digits of the binary representation of `x` have been used up. To obtain longer sequences we define `x` as a bigfloat with enough nonzero digits. Notice that because of the methods we defined earlier, `bernoulli` will work for bigfloats with no change:

```
> x <- 1/pi
> bern1 <- bernoulli(x, 65)
> x <- as.bigfloat(Math('N[1/Pi,100]'))
> bern2 <- bernoulli(x, 65)
```

This example produces two sequences, `bern1` and `bern2`, both starting at `1/pi` but with different precision. The resulting sequences are plotted in Figure 2 which shows a noticeable difference after 50 iterations. After 55 iterations `bern1` becomes zero.

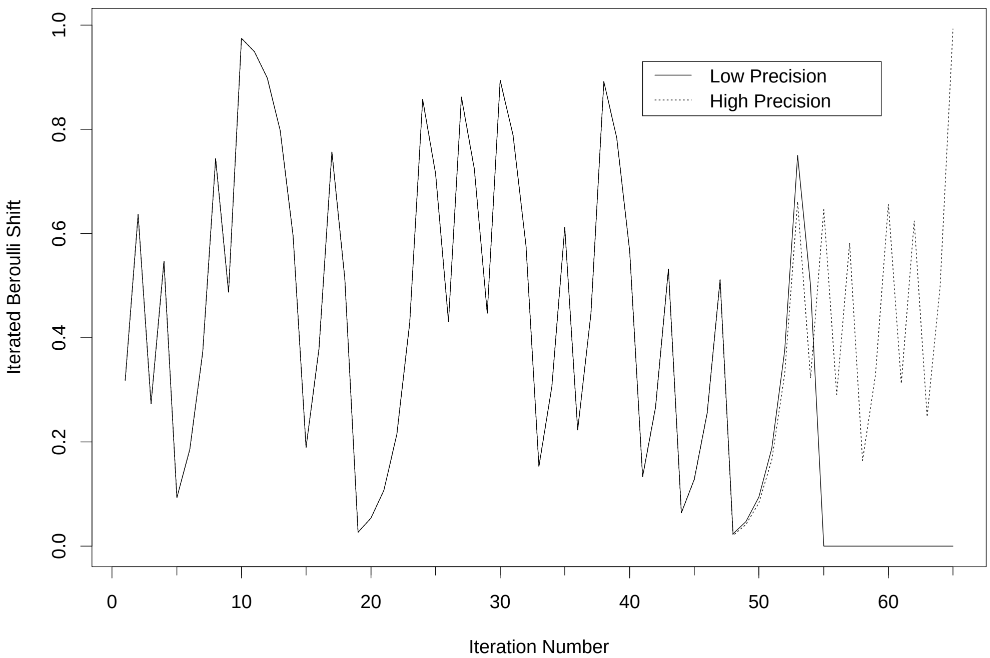

Figure 2. Bernoulli Shift Sequences for Two Precisions

References

Cabrera, Javier, (1989), *Some experiments with maximum likelihood estimation using symbolic manipulations*, Proceedings of the 21st symposium on the interface, editor K. Berk.

Becker, Richard A., John M. Chambers, Allan R. Wilks (1988), *The New S Language*, Wadsworth, Pacific Grove, California.

Chambers, John M., Trevor Hastie, eds. (1991) *Statistical Models in S*, Wadsworth, Pacific Grove, California.

McCullagh, P., Nelder, J.A.(1983), *Generalized linear models*, Chapman and Hall, New York.

Tukey, John W. (1977), *Exploratory Data Analysis*, Addison-Wesley, Reading Massachusettes.

Wolfram, Stephen (1988), *Mathematica: A System for Doing Mathematics by Computer*, Addison-Wesley, Reading, Massachusettes.

Listing 1

```
as.math <- function(expr)
{
      m <- mode(expr)
      switch(m,
      NULL = "Null",
      logical = c("False","True")[1+expr],
      numeric = Math.numbers(expr),
      complex = paste("Plus[", as.math(Re(x)), ",I", as.math(Im(x)), "]"),
      character = expr,
      expression = as.math(expr[[1]]),
      name = switch(expr,
            pi = "Pi",
            T = "True",
            TRUE = "True",
            F = "False",
            FALSE = "False",
            NULL = "Null",
            Math.names(expr)
            ),
      "(" = as.math(expr[[2]]),
      "{" = paste("(", paste(unlist(lapply(as.list(expr), as.math)),
              sep = "", collapse = ";"), ")", sep = ""),
      { # all other modes
            if(m == "call") {
                  m <- as.character(expr[[1]])
                  if(m == "-" & length(expr) == 2)
                  m <- "uminus"
                  expr <- expr[-1]
            }
            args <- comma.sep(paste(unlist(lapply(expr, as.math))))
            paste(Math.functions[m], "[", args, "]", collapse = "")
      }
      )
}
```

Listing 2

```
D <- function(expr, vars)
{
     expr <- substitute(expr)
     if(mode(expr) == "name" && exists(as.character(expr), mode="call"))
          expr <- eval(expr)
     expr.names <- all.objects(expr, vars)
     math.assign(expr.names)
     value <- sform(paste("D[", as.math(expr), ",", comma.sep(vars), "]"))
     math.remove(expr.names)
     value
}
```

Listing 3

```
Integral <- function(expr, vars, lims, numeric=!missing(lims))
{
     expr <- substitute(expr)
     if(mode(expr) == "name" && exists(as.character(expr), mode="call"))
          expr <- eval(expr)
     if(!missing(lims)) {
          lims <- as.matrix(lims)
          if(dim(lims)[2] != 2)
               stop("lims must be a 2-vector or a 2-column matrix")
          vars <- Math.array(c(vars, lims))
     }
     math.fcn <- if(numeric) "NIntegrate" else "Integrate"
     expr.names <- all.objects(expr, vars)
     math.assign(expr.names)
     value <- sform(paste(math.fcn, "[", as.math(expr), ",", comma.sep(vars), "]"))
     math.remove(expr.names)
     value
}
```

Listing 4

```
MLEiid <- function(df, vars, theta, data, init)
{
     df <- substitute(df)
     if(mode(df) == "name" && exists(as.character(df), mode="call"))
          df <- eval(df)
     expr.names <- all.objects(df, c(vars, theta, "data"))
     math.assign(expr.names)

     # define sample in Mathematica
     math.assign("data", list(data)))

     # define density in Mathematica
     Math(paste("df[", vars, "_,", theta, "_]:=", as.math(df), ";"))

     # define log likelihood in Mathematica
     Math("ll[f_,x_,theta_]:=Sum[Log[f[x[[i]],theta]],{i,1,Length[x]}];")

     # construct and execute the call to FindMinumum[]
     fun <- "-ll[df, data, theta]"
     wrt <- paste("{theta,", init, "}")
     right.piece <- "[[2]][[1]][[2]]"
     value <- sform(paste("FindMinimum[", fun, ",", wrt, "]", right.piece))

     Math(math.remove(c(expr.names, "data", "df", "ll")))
     value
}
```

Listing 5

```
logist <-
function(x,y)
{
      x <- cbind(1,x)
      beta <- paste("beta", 1:ncol(x), sep = "")
      expr.names <- c("x","y","beta")
      math.assign(expr.names,list(x,y,beta))

      # define the log-likelihood in Mathematica
      lik.expr <- as.math(substitute(lik <- -y%*%(x%*%beta) +
                sum(log(1+ exp(x%*%beta)))))

      # establish the likelihood equations ...
      Math(lik.expr)
      eqns   <- Math.list( paste("D[lik,",beta,"]==0") )
      params <- comma.sep( paste("{", beta,",0}") )

      # ... and solve them!
      value  <- sform(paste("FindRoot[",eqns,",",params,"]",sep=""))
      math.remove(c(expr.names,"lik"))
      return(as.numeric(unlist(parse(text=value))[(1:ncol(x))*3+1]))
}
```