R-Matlab Interface

Bitao Liu
Duncan Temple Lang*

January 15, 2007

Abstract

This is a *very early draft* of documentation discussing the RMatlab interface.

Matlab and R are two interactive, high-level programming languages used in scientific computing. The languages have a lot in common but have very different target audiences and focii. R is primarily used by the statistical community for advanced data analysis and research in statistical methodology; Matlab is primarily used by engineers. The functionality of the two languages does overlap, but also complement each other very well. R has little suppport for image processing, differential equations, and so on, while Matlab has only limited support for modern statistical methodology and stochastic data analysis. Of course, both languages provide the primitives to develop such facilities, being general purpose programming languages, but it makes much more sense to reuse existing, specialized code when it is available. Rather than requiring Matlab users to program in R to access statistical functionality, and R users to write Matlab code for engineering-oriented tasks, we present an interface that allows one to call R from Matlab and vice versa. This allows each group to transparently leverage the functionality provided by the other. We describe the `RMatlab` package which provides this inter-system interface. The model is quite simple and general and mimics that used in other Omegahat inter-system packages (see SJava, RSPerl, RSPython, ROctave, RXLisp, RinS).

1 Introduction

As we mention in the abstract, Matlab and R share many similarities, but are quite different in many ways. The S language – of which R is a dialect – is quite rich both in terms of statistical data analysis and numerical computation, and also software development. The R environment provides advanced conceptual but common facilities used by statisticians in research and data analsysis. It allows statisticians to think about computations in much the same way as they do the underlying statistical methodology and data analysis. Records, data frames, subsetting, formulae, models are aspects that make R uniquely statistical in nature. Matlab is significantly more oriented towards interactive engineering tasks. The language is more foucsed on direct matrix calculations and providing the primitives for numerical computing. That language provides the essential data structures, adding to its simplicity, sometimes at the expense of its expressiveness. Matlab focuses on computations such as eigen-value.

One of the motivations of this package (and other inter-system packages) is to allow scientists to program their computational tasks quickly and easily and using well-tested and reliable software. When possible, reusing existing software that has been used extensively by others is significantly better than writing our own. And avoiding the overhead of writing such software and distracting ourselves from the original task allows us to focus on expressing a computation directly and succinctly.

2 The Interface

The `RMatlab` package provides a way for R to call Matlab functions, and Matlab to call R functions. We can start Matlab from within R, or we can embed R within Matlab. When we do the latter (starting R within Matlab) using the `RMatlab` package, we start an R interpreter as MEX-accessed C code within the Matlab process. This allows us to call R functions from within the Matlab process using the same address space. This makes the inter-system communication very fast and allows objects to be shared directly by reference. And since R is embedded within

*Please send correspondence to duncan@wald.ucdavis.edu.

Matlab, it can make calls back to Matlab. This allows for a range of intresting computations, and ensures that we can use the two languages in tandem and program in the most convenient environment for individual tasks.

Calling R from Matlab can occur in two ways. As we have just discussed, if R is started from within a Matlab session, R functions invoked from Matlab can call back to Matlab functions. This is called the MEX interface. The two interpreters share the same single process space and the communication is direct in the same way that Matlab calls MEX files and other external interfaces.

The second way that R can access Matlab is by starting a separate Matlab process and sending commands to it. This is the Engine Interface to Matlab. (This same mechanism can be used for connecting Matlab to R on Windows using the DCOM server interfaces provided by R.) The fact that this is a separate process makes the communication slower but allows us to have the process on a different machine. This allows us to do a form of distributed computing. Unfortunately, the Engine API provides only a limited set of routines by which we can access the Matlab interpreter. They suffice to allow us to hide the differences between the MEX and Engine interface, but the implementation details do have consequences for the Engine interface. We will discuss these below.

3 Calling R from Matlab

We provide 3 simple functions in Matlab to access functionality in R. The first is used to initialize the R engine and the other two are used to invoke arbitrary R functions. The functions are *initializeR* *callR* and *Matlab*callNamedR. Using these, one has access to most all of R.

3.1 initializeR

It seems natural to expect that one must start the R interpreter before calling any R functions. And we do this from within Matlab using the function *initializeR*. (The package needs to ensure that this is done before calling an R function.) The inputs to *initializeR* should consist of a Matlab "cell" giving the command line arguments to R as strings. We note that the first element of these command line arguments should be a name to use for the program; we use `RMatlab`. It currently has no consequence. The remaining arguments are the regular ones that we would provide to R on the command line when starting R from a shell.

An example will make this more concrete. The following creates an instance of the R interpreter with the Matlab process.

```
initializeR({'Rmatlab' '--no-save' '--no-restore'})
```

This tells R not to read an .RData file, if it exists, in the working directory on startup and to not save the global environment/workspace when terminating R.

Of course, we can create the argument cell structure in a separate step and store them in a Matlab variable and then pass them to the *initializeR* call. For example,

```
args = {'Rmatlab' '--no-save' '--no-restore'}
```

or

```
args = cellstr(char('Rmatlab', '--vanilla'))
```

creates the variable `args`. Then, we can call thes initialization routine to start R with these arguments.

```
initializeR(args)
```

The reason for using cells rather than strings is to avoid padding the string elements with blanks. In Matlab,

```
 [ 'RMatlab' ; '--no-save']
```

gives an error because the strings don't have the same length. Unfortunately, Matlab stores its string "vectors" as a matrix and so each string (i.e. element in the vector) must have the same number of characters. The function *char* avoids the error and is a convenient way to create a Matlab object storing the strings:

```
 char('RMatlab', '--no-save')
```

Again, this creates a matrix whose dimensions are 1 by 8; in other words, the RMatlab is extended with an extra space to make it the same length as `--no-save`. Accordingly, the inputs cannot be used directly as command line arguments and we would have to trim the arguments. The result is that we use a cell array to represent the command line arguments as is, without the extra padding.

Note that we can easily define a M-function in Matlab to provide a more convenient interface to this *initializeR* primitive. The function

```
function status = startR(varargin{:})
  status = initializeR([ {'RMatlab'}, varargin ] )
end
```

merely avoids the need to remember to specify the `RMatlab` as the first command line argument. This is a small advantage!

3.2 callR

The next and most important function/routine in the Matlab-to-R is *callR*. This is the mechanism with which we can invoke an R function from within Matlab. This is the fundamental building block with which we can gain access to almost everything in R.

The function is quite straightforward. The first argument is the name of the R function to be called and this is required. The remaining arguments are the values to be passed (in the same order) as arguments to the R function. These arguments are marshalled to the R engine using the conversion mechanism discussed below. The result of the R function is converted back to a Matlab value and returned as the result of the *callR* function.

Let's start with a very simple example of generating random numbers using R's RNG facilities. We generate 10 random values from the Poisson distribution with mean 3 using the call

```
o = callR('rpois', 10, 3)
```

This is equivalent to evaluating the R expression

```
rpois(10, 3)
```

The values 10 and 3 are converted to the corresponding values in R (numeric vectors of length 1) and then passed in the call to *rpois()*. The resulting R integer vector of length 10 containing the random numbers is converted to a 10 by 1 Matlab array and this is assigned to the variable `o`. We can use this directly within Matlab in subsequent commands without any reference to R and where the data come from. For example,

```
mean(o)
```

Arbitrary R functions can be called using *callR* using the name of the function. Clearly, all that is needed is that the Matlab values are marhsalled across to the R call and the result returned appropriately. Soon, we will be able to invoke anonymous functions that are returned as results of earlier calls.

3.3 Named Arguments in R Function Calls

The function *rnorm()*, for example, illustrates a problem with the *callR*. Suppose we want to generate numbers from a $N(0, 9)$ distribution, i.e. with an SD of 9. In R, we can use named arguments as in

```
rnorm(10, sd = 9)
```

This allows us to omit the value of the mean and use the default value for that argument. In other R functions, the fact that an argument is missing rather than specified with the default value can be important. So we need to be able to call R functions with explicit names for the arguments. Unfortunately, given my current and limited knowledge of Matlab, there are no named arguments in Matlab. Instead, people use conventions to associate names with values. Without this as a fundamental part of the language, the conventions become akward. For example, suppose we want to call an R function

```
bar =
function(x, y, xlab = "X", ylab = "Y", title = "A Plot")
{
  ...
}
```

and pass it vectors for the first two parameters and a value for the *title* parameter. We could use something like the following:

```
 callR('bar',
        'x', [1 2 3 4], 'y', [10, 9, 8, 7], 'title', 'My plot')
```

Here we identify *each* argument by name. This is not a bad thing as it makes the call explicit and avoids any ambiguity. The downside is that we *must* specify the arguments even when the position would make some of them clear.

In R, we would specify this call as

```
bar( c(1, 2, 3, 4), c(10, 9, 8, 7), title = 'My plot')
```

We can leverage this approach in the Matlab interface using a differnt function, say *callNamedR* in which we specify the arguments as

```
callNamedR('bar',
           [1 2 3 4], [10, 9, 8, 7],
           {'title', 'My plot'})
```

The idea is that unnamed arguments are given as regular arguments and the named arguments are given as a Matlab cell consisting of name-value pairs.

To call *rnorm()* in our initial example above, we can now use

```
callNamedR('rnorm', 10, {'sd', 9})
```

3.4 Evaluating R commands

Many users who think about calling R from Matlab or vice verse will be inclined to send an R command from Matlab. If we want to generate 10 values from a $N(0, 9)$ as above, we could evaluate the command

```
 rnorm(10, sd = 9)
```

We have provided an R function named *.REvalString()* in the RMatlab package which can be used to evaluate R commands all parcelled up as a string. The result is returned directly to the caller and so this can be called in Matlab as

```
data = callR('.REvalString', 'rnorm(10, sd = 9)')
```

This is convenient for simple use. Unfortunately, it does not work well for more general situations. We have to construct the command as a string. When we can type the inputs as a literal string (as above), this is fine. When we have to merge values that are stored in variables, this becomes more tedious. For example, if we have a Matlab function that takes the mean and sd as arguments, we have to concatenate the values into the command string. This might be done with the Matlab command

```
 strcat('rnorm(', num2str(n), ', sd = ', num2str(sd), ')')
```

This is certainly less elegant than a direct call to *rnorm()*. It gets worse in two other ways. If we allow different types for the inputs (e.g. numbers, strings, cells, etc.), we will need a more general mechanism to convert the values to strings. And secondly, we have to be careful with quotation marks in strings. We must escape quotes within the command.

4 Basic Examples

5 Advanced Examples

Ode.

Fast matrix operations.

Graphics.

Round-tripping to ensure that we get what we expect.

x = callR(".MatlabMexCall", "matrix", 2, 2)

6 Calling Matlab from R

Calling Matlab from R turns out to be a little more limited than the other way around. This is primarily because the folks at MathWorks have apparently provided a more traditional model for interacting with the Matlab interpreter or "Engine". This model allows only for evaluating Matlab commands presented as strings and setting and retrieving Matlab variables in the work space. We can use this to implement a reasonably powerful connection between R and Matlab, but

it is somewhat limited and more akward than being able to call Matlab functions directly and anonymously.

The `RMatlab` provides functions for the following.

.MatlabInit starting the Matlab engine,

.MatlabEval evaluating a Matlab command,

.MatlabGet retrieving the value of a variable in the Matlab workspace,

.MatlabPut setting the value of a variable in the Matlab workspace.

.MatalClose closing the Matlab engine.

These primitives allow us to construct Matlab commands in R and to fetch the results. We must start by initializing the Matlab engine. We can pass different command line options to the startup just as we would when invking Matlab from the Unix shell. (On Windows, these are ignored.) Rather than invoking the matlab executable, we specify "0" as documented in the Matlab manual[**?**].

With the engine initialized, we can evaluate Matlab commands.

Our model when calling R from Matlab was to call a function by name and pass the values directly from Matlab to R, e.g.

```
x = callR("foo", 1:10, {"a", "bcde"}, rand(4, 3))
```

If we had a reference to an anonymous (un-named) R function, then we would be able to invoke that also with the same mechanism. The primitives above do not allow us to use the same model directly. However, with a few drawbacks, we can hide the details from the user. We would like to be able to call a Matlab function from R in much the same way as above, something like

```
x = .Matlab("foo", 1:10, c("a", "bcde"), matrix(rnorm(12), 4, 3))
```

If we were to use the *.MatlabEval()* function directly, we would have to write each of the arguments in its Matlab format with the values currently in the R arguments. This would be something like

```
"foo([1 2 3 4 5 6 7 8 9 10], {\"a\", \"bcde\"}, [
```

In other words, we have to serialize each R object in a form suitable for constructing the corresponding value in Matlab. For a matrix, we would need a function something like

```
paste("[", paste(apply(m, 1, paste, collapse = " "), collapse = "; "), "]")
```

Of course, this string representation loses precision in the numbers by limiting the decimal places, although we can control this. And we have to be careful to escape the quotes for character matrices. All in all, this is quite ugly and not very general. What about transferring R functions as arguments, or a model formula, or a connection object. Well of course, these don't have a corresponding form in Matlab that is useful. But we often want to pass these in a function call as arguments to another function call which might be in R.

Armed with the *.MatlabPut()* and *.MatlabGet()* functions, we can provide a more flexible interface that can potentially grow to accomodate a richer interface. Rather than constructing a Matlab command by serializing the R arguments as strings, we can first marshall each value to Matlab and assign it to a variable in the Matlab workspace using *.MatlabPut()*. Then, we can construct a simpler Matlab command to call the function of interest referencing these named variables and assign the result to a particular Matlab variable. Then, we can marshal that value back to R via the *.MatlabGet()* function. Our *.Matlab()* function is now more[1] elegantly written as

```
.Matlab =
function(funcName, ..., engine)
{
  args <- list(...)
  names(args) <- paste("r_arg", 1:length(args), sep = "_")
  .MatlabPut(.values = args, engine = engine)

  on.exit(.MatlabRemove(names(args), engine = engine))

  cmd = paste("r_result = ", funcName, "(",
                paste(names(args), collapse=", "),
              ");")
  .MatlabEval(cmd, engine = engine)

  ans = .MatlabGet("r_result", engine = engine)

  .MatlabRemove("r_result", engine = engine)
  ans
}
```

[2]

What is missing in this interface? The ability to use the mexCallMATLAB would be useful. The implementation above uses global variables within the Matlab workspace. This could cause problems.

It might be possible to be able to call mexCallMATLAB in the restricted case of calling Matlab from R from within Matlab. In other words, this may require that we initiate the original communication from Matlab but callback to Matlab from R.

The idea is quite simple. From within Matlab, we start R. Then we call R to load the RMatlab package to load the R functions to access Matlab and the associated C code in RMatlab.so:

```
callR('library', 'RMatlab')
```

Then we can use the following

```
o = callR('.MatlabMexCall', 'rand', 1, 1)
```

to a very basic computation. We start from Matlab and call the R function *.MatlabMexCall()*. We pass it 3 argument - the name of a Matlab function to call and arguments (1 & 1) to be passed to that Matlab function. R evaluates the call to *.MatlabMexCall()* and accordingly invokes the specified Matlab function 'rand'. Matlab returns the answer to R and R returns this to the original Matlab invocation of callR. This is of course a simple, artificial example. The idea is that the R function we call from Matlab can be quite complicated and can make calls to Matlab functions during its execution.

[1] but still not actually elegant

[2] The version in the R package is slightly more sophisticated than this, but only in dealing with names of R arguments.

6.1 Example: Optimization

We now move to a more advanced example in which we use Matlab's optimization facilities to find the maximum of a mathematical function defined in R. We might be interested in doing this if we think Matlab's optimization methods are more appropriate for the function to be optimized or if we merely want to validate that we get the same answer. We might also want to compare the number of iterations it takes to reach the optimial value.

For our example, consider a simple log-likelihood function that has two parameters μ and σ and associated observed data. Let's denote this as this as $L(\mu, \sigma | X_1, \ldots, X_n)$. We can take advantage or R's concept of a closure to combine the observed data with the particular instance of the function. Let's use a Normal density function, so our log-likelihood can be written in R as

```
NormalLikelihood =
function(x)
{
 n = length(x)
 S2 = sum(x^2)
 S = sum(x)

    # XXX
    # Assume parameters are a single vector, not separate arguments.
    # Nicer as two arguments. Check Matlab interface to optimization
    # functionality.
 function(theta) {
    -n*log(sqrt(2*pi*theta[2])) + (S2 - 2*S*theta[1] + theta[1]^2)/theta[2]
 }
}
```

We also want the derivative, that is the Hessian, of the function.

```
data = rnorm(100)
l = NormalLikelihood(data)
ans = .Matlab("optim", l, h, c(0, 1) )
```

(We expect) The outputs are returned as a single row vector containing estimates of μ and σ.

When we call the Matlab function *optim* from R, the Matlab engine takes over control and evaluates that function call until it terminates (normally or abnormally).

It evaluates the function to be optimized at different points. Each time it does this, it passes control back to R which evaluates the function call with inputs from Matlab. As this proceeds, data is sent from Matlab to R and back to Matlab with the result of the evaluation of the operand at this point in the parameter space. Eventually, the Matlab optimizer converges (or not) and terminates and returns the result to R and yields control back to R. If we put a simple statement to produce output in our likelihood function, we can monitor the different inputs that are passed from Matlab and even display them in a plot. (Note that the display will not be updated because of event loop issues.)

Can this be done? What needs to feasible in our interface and hence in Matlab? We need to be able to pass an R function as a "reference" to Matlab. We need to be able to identify that object as an R function so that we can invoke it. And we need to be able to resolve the R reference back on the R side when identified in Matlab. Ideally, we would be able to extend the Matlab data structures to be able to represent an R function as a "callbable" Matlab value and the low-level, internal evaluation method would merely be call to the R function. If this is not possible, we should be able to dynamically create a Matlab function within the Matlab session that is callable and whose body is an invocation of the associated R function:

```
f  = function(...)
{
  callR(ref, ...) # get the number of arguments from the call to f.
}
```

The issue here is how to represent the 'ref' value to be able to identify the R function object.

References

[1] The MathWorks, *Matlab external interfaces*, URI: http://www.mathworks.com/access/helpdesk/help/techdoc/apiref/apiref.html, November 2004.

[2] Andrew Sterian, *Pymat developers page*, URI: http://sourceforge.net/projects/pymat, October 2004.