An Introduction to the Interactive Debugging Tools in R

Roger D. Peng
UCLA Department of Statistics

August 28, 2002

1 Introduction

The purpose of this document is to provide a brief introduction to the built-in program debugging tools in the R statistical computing environment. The five functions that will mainly be covered are `traceback`, `debug`, `browser`, `trace`, and `recover`. Throughout, the `typewriter font` will be used to indicate R code.

It is important to note that debugging is a practice which gets considerably easier as one's familiarity with the language increases. In some ways it can be more "art" than "science". For example, knowing where to look in a 500 line program after it has just halted execution is sometimes just a "feeling" one develops after much previous suffering.

2 Trouble with your Droids?

R has a very rich language with which users can write very useful but potentially complex functions. As with programs written in any other language, functions written in R can contain unforseen problems which lead to failure. The purpose of the debugging tools is to help the programmer find these problems quickly and efficiently. R is a generally friendly language and the problems which arise are typically unlike those found in a language such as C or C++. For example, in C, memory management can be the source of the most malicious bugs; in R memory management is not even an issue.

The R system has two main ways of reporting a problem in executing a function. One is a *warning* while the other is a simple *error*. The main difference between the two is that warnings do not halt execution of the function. The purpose of the warning is to tell the user "Something unusual happened during the execution of this function, but the function was nevertheless able to execute to completion." One example of getting a warning is when you take the log of a negative number:

```
> log(-1)
[1] NaN
Warning message:
```

```
NaNs produced in: log(x)
```

Here, the `log` function returned a value (in this case, it returned `NaN`), but produced a warning also.

An error is usually a problem that is fatal and results in a complete halt in execution of the function. That is, because of some problem in the function, the function simply cannot execute to completion. Consider the following function:

```
message <- function(x) {
  if(x > 0)
    print(``Hello'')
  else
    print(``Goodbye'')
}
```

This function simply prints “Hello” or “Goodbye” depending on whether `x` is greater than or less than 0. On the surface, it would seem that this function should never fail (!). However, observe this R session:

```
> x <- log(-1)
Warning message:
NaNs produced in: log(x)
> message(x)
Error in if (x > 0) { : missing value where logical needed
```

Notice, that the first operation (taking the log) only produced a warning, but passing the resulting value of `x` into `message` ended in a fatal error. The `message` function died at the `if` statement. The reason is that value of `x` was `NaN`, which was the result of taking the log of -1. `NaN` is a special value used to indicate undefined operations. For example, $0/0$ is `NaN`. When `x` is passed to `message`, `message` tries to compare `NaN` with 0, which is clearly undefined. Since the function doesn't specify what to do in this situation, the execution must stop.

Changing the value of `x` here “fixes” the problem:

```
> x <- 4
> message(x)
[1] "Hello"
```

However, the problem is not really fixed because there is nothing stopping another user (or even yourself!) from passing `log(-1)` into `message` at some later date. Writing robust code (i.e. code which checks for imput errors) is important for larger programs, but since it is a general problem of programming, it will not be discussed here.

3 Debugging Tools

3.1 Printing the Call Stack with `traceback`

In Section 2 we showed some possible messages that can result from a buggy program. What can you do when your program produces such messages? For now, let's assume you've received a fatal error, not a warning. The first thing you might want to do is print the call stack, i.e. print the sequence of function calls which led to the error. This can be done using the `traceback` function. The `traceback` function prints the list of functions which were called before the error occurred. This can be uninteresting if the error occurred at a top level function. For example, recall the `message` function from Section 2. An example of the usage of `traceback` is in the following R session:

```
> message(log(-1))
Error in if (x > 0) { : missing value where logical needed
In addition: Warning message:
NaNs produced in: log(x)
> traceback()
1: message(log(-1))
```

Here, traceback shows in which function the error occurred. However, since only one function was in fact called, this information is not very useful. It's clear that the error occurred in the `message` function. Now, consider the following function definitions:

```
f <- function(x) {
    r <- x - g(x)
    r
}

g <- function(y) {
    r <- y * h(y)
    r
}

h <- function(z) {
    r <- log(z)
    if (r < 10)
        r^2
    else r^3
}
```

Suppose the function we are interested in is `f`. In this case, `f` calls `g`, which calls `h`, and then returns a result. You should recognize immediately that there may be a problem with the `if` statement in the `h` function. Here is where traceback may be useful:

```
> f(-1)
Error in if (r < 10) r^2 else r^3 : missing value where logical needed
In addition: Warning message:
NaNs produced in: log(x)
```

What happened here? First, the function `f` was halted somewhere because of a bug. Furthermore, we got a warning from taking the log of a negative number. However, it's not immediately clear *where* the error occurred during the execution. Did `f` fail at the top level or at some lower level function? Upon receiving this error, we could immediately run `traceback` to find out:

```
> traceback()
3: h(y)
2: g(x)
1: f(-1)
```

`traceback` prints the sequence of function calls in reverse order from the top. So here, the function on the bottom, `f`, was called first, then `g`, then `h`. From the `traceback` output, we can see that the error occurred in `h` and not in `f` or `g`.

3.2 Stepping Through with `debug`

Sometimes you want to interact with your function while it is running to get a better idea of the function's behavior. In the example in Section 3.1, we used `traceback` to figure out where in the call stack an error occurred. However, `traceback` doesn't tell you where in the *function* the error occurred. We still need to do some work.

In the example in Section 3.1 with `f`, `g`, and `h`, using `traceback` was probably enough because we could immediately go back into our code and just "figure out" where the problem was. However, in most useful functions, it may not be clear where the problems are and we might want to *step through* the function line-by-line to identify the specific location of a bug. To do this we use the `debug` function.

`debug` takes a single argument — the name of a function. When you pass the name of a function to `debug`, that function is *flagged for debugging.* In order to unflag a function, there is the corresponding `undebug` function. When a function is flagged for debugging, it does not execute on the usual way. Rather, each statement in the function is executed one at a time and the user can control when each statement gets executed. After a statement is executed, the function suspends and the user is free to interact with the environment. This kind of functionality is what most programmers refer to as "using the debugger" in other languages.

For the next example, we will set up the problem of estimating a Normal mean by minimizing the sum of squared differences. Given a sample $x_1, \ldots, x_n$, we need to compute

$$SS = \sum_{i=1}^{n} (x_i - \mu)^2 .$$

First, we define the following function in R:

```
SS <- function(mu, x) {
    d <- x - mu
    d2 <- d^2
    ss <- sum(d2)
    ss
}
```

The function SS simply computes the sum of squares. It is written here in a rather drawn out fashion for demonstration purposes only. Now we generate a Normal random sample:

```
> set.seed(100)  ## set the RNG seed so that the results are reproducible
> x <- rnorm(100)
```

Here, x contains 100 Normal random deviates with (population) mean 0 and variance 1. We can run SS to compute the sum of squares for x and a given value of mu. For example,

```
> SS(1, x)
[1] 208.1661
```

But suppose we wanted to interact with SS and see how it operates line by line. We need to flag SS for debugging:

```
> debug(SS)
```

The following R session shows how SS runs in the debugger:

```
> SS(1, x)
debugging in: SS(1, x)
debug: {
    d <- x - mu
    d2 <- d^2
    ss <- sum(d2)
    ss
}
Browse[1]> n
debug: d <- x - mu
Browse[1]> n
debug: d2 <- d^2
Browse[1]> n
debug: ss <- sum(d2)
Browse[1]> n
debug: ss
Browse[1]> n
exiting from: SS(1, x)
[1] 208.1661
```

The first thing that happens when you execute a function in the debugger is the body of the function is printed. Nothing is actually executed, just the code is printed to the screen. After the function body is printed you are confronted with the following prompt:

```
Browse[1]>
```

You are now in what is called the "browser". Here you can enter one of four basic debug commands. Typing `n` executes the current line and prints the next one. At the very beginning of a function there is nothing to execute so typing `n` just prints the first line of code. Typing `c` executes the rest of the function without stopping and causes the function to return. This is useful if you are done debugging in the middle of a function and don't want to step throught the rest of the lines. Typing `Q` quits debugging and completely halts execution of the function. Finally, you can type `where` to show where you are in the function call stack. This is much like running a `traceback` in the debugger (but not quite the same).

Besides the four basic debugging commands mentioned above, you can also type other relevant commands. For example, typing `ls()` will show all objects in the local environment. You can also make assignments and create new objects while in the debugger. Of course, any new objects created in the local environment will disappear when the debugger finishes. If you want to inspect the value of a particular object in the local environment, you can print its value, either by using `print` or by simply typing the name of the object and hitting return. If you have objects in your environment with the names `n`, `c`, or `Q`, then you must explicitly use the `print` function to print their values (i.e. `print(n)` or `print(c)`).

Here's another session with `SS`:

```
> SS(2, x)
debugging in: SS(2, x)
debug: {
    d <- x - mu
    d2 <- d^2
    ss <- sum(d2)
    ss
}
Browse[1]> n
debug: d <- x - mu
Browse[1]> d[1]  ## Print the value of first element of d
[1] -0.4856523
Browse[1]> n
debug: d2 <- d^2
Browse[1]> hist(d2) ## Make a histogram (not shown)
Browse[1]> n
debug: ss <- sum(d2)
Browse[1]> n
debug: ss
```

```
Browse[1]> print(ss)  ## Show value of ss; using print() is optional here
[1] 503.814
Browse[1]> ls()
[1] "d"  "d2" "mu" "ss" "x"
Browse[1]> where
where 1: SS(2, x)

Browse[1]> y <- x^2  ## Create new object
Browse[1]> ls()
[1] "d"  "d2" "mu" "ss" "x"  "y"
Browse[1]> y
  [1] 2.293249e+00 1.043871e+00 5.158531e-01 3.677514e-01 1.658905e+00
  [... omitted ...]
Browse[1]> c  ## Execute rest of function without stepping
exiting from: SS(2, x)
[1] 503.814
> undebug(SS)  ## Remove debugging flag for SS
```

One last thing worth mentioning is that you can flag functions for debugging while you are debugging another function. That is, you do not need to flag every function you want to debug before you execute the top level function. You can set debugging flags on the fly. For example, with the SS examle above, we could have flagged the sum function for debugging while we were debugging SS (this would have been rather uninteresting, though). Of course, you must flag the function for debugging *before* it is called in the code. One final example:

```
> debug(SS)
> SS(2, x)
debugging in: SS(2, x)
debug: {
    d <- x - mu
    d2 <- d^2
    ss <- sum(d2)
    ss
}
Browse[1]> n
debug: d <- x - mu
Browse[1]> n
debug: d2 <- d^2
Browse[1]> n
debug: ss <- sum(d2)
Browse[1]> debug(sum)  ## Flag sum for debugging
Browse[1]> n
debugging in: sum(d2)
```

```
debug: .Internal(sum(..., na.rm = na.rm))
Browse[1]> where  ## Print the call stack; there are 2 levels now
where 1: sum(d2)
where 2: SS(2, x)

Browse[1]> n
exiting from: sum(d2)
debug: ss
Browse[1]> n
exiting from: SS(2, x)
[1] 503.814
> undebug(SS); undebug(sum)
```

3.2.1 Explicit Calls to `browser`

It is possible to do a kind of "manual debugging" if you don't feel like stepping through a function line by line. The function `browser` can be used to suspend execution of a function so that the user can browse the local environment. Suppose we edited the `SS` function from above to look like:

```
SS <- function(mu, x) {
    d <- x - mu
    d2 <- d^2
    browser()
    ss <- sum(d2)
    ss
}
```

Now, when the function reaches the third statement in the program, execution will suspend and you will get a `Browse[1]>` prompt, much like in the debugger.

```
> SS(2, x)
Called from: SS(2, x)
Browse[1]> ls()
[1] "d"  "d2" "mu" "x"
Browse[1]> print(mu)
[1] 2
Browse[1]> mean(x)
[1] 0.02176075
Browse[1]> n
debug: ss <- sum(d2)
Browse[1]> c
[1] 503.814
```

Notice that the first two lines in the function were not printed. This kind of use of `browser` can be useful if you have a vague idea as to where a bug may be in your program.

3.3 Inserting Code with `trace`

The `trace` function is very useful for making minor modifications to functions "on the fly" without having to modify functions and re-sourcing them. It is especially useful if you need to track down an error which occurs in a base function. Since base functions cannot be edited by the user, `trace` may be the only option available for making modifications.

Note that `trace` has an extensive help page which should be read in its entirety. We will try to summarize the highlights here.

The example we will use here is fitting a point process model via maximum likelihood. The point process model here is specified by it's conditional intensity, $\lambda(t;\mu)$, where μ is the unknown parameter we want to estimate. For this example our model will be

$$\lambda(t) = \mu t.$$

The log-likelihood of the model is

$$\ell(\mu) = \sum_{i=1}^{n} \log \lambda(t_i;\mu) - \int_0^T \lambda(s;\mu)\,ds$$

where n is the number of events we observe and T is the upper time boundary over which we observe the events. Rather than maximize this log-likelihood, we will minimize the negative log-likelihood; this is slightly more natural to do in R.

First we will simulate a point process over $[0,1]$ according to the given model:

```
> set.seed(100)
> p <- sort(runif(200))
> thin <- rbinom(200, 1, p)
> pp <- p[thin == 1]
> hist(pp, nclass = 20)  ## Not shown
```

Now we write out the negative log-likelihood function which takes two arguments: `mu` the unknown parameter and `x` the vector of event times.

```
nLL <- function(mu, x) {
    z <- mu * x
    lz <- log(z)
    L1 <- sum(lz)
    L2 <- mu/2
    LL <- -(L1 - L2)
    LL
}
```

In order to minimize the negative log-likelihood, we will have to pass `nLL` to an optimization routine. In R, there are two such routines: `nlm` and `optim`. Here we will use `optim`, which allows the user to choose from a variety of different optimization procedures. `optim` requires a starting value, the function to be optimized, the method for optimization and other arguments that are necessary for the function to be optimized (typically data). In this example, we use a starting value (albeit not a wise one) of $100,000$ and a quasi-Newton optimization procedure due to Broyden, Fletcher, Goldfarb, and Shanno.

```
> optim(100000, nLL, method = "BFGS", x = pp)
$par
[1] 188.0121

$value
[1] -365.7472

$counts
function gradient
      41       19

$convergence
[1] 0

There were 21 warnings (use warnings() to see them)
```

Here we see the optimizer does converge on the value of 188.0121. However, there were warnings set off during the optimization and we can use `warnings` to view them.

```
> warnings()
Warning messages:
1: NaNs produced in: log(x)
2: NaNs produced in: log(x)
3: NaNs produced in: log(x)
[... omitted ...]
21: NaNs produced in: log(x)
```

All of the warnings come from the `log` function which is of course used in the negative log-likelihood function. In this situation, using `debug` would be problematic because as the original `optim` output shows, the optimizer makes 41 calls to the `nLL` function. However, there were only 21 warnings, so some of the calls to `nLL` were fine and did not cause a warning. It would be tedious to have to step through the `nLL` function line by line 41 times. In larger optimization problems this could be hundreds of function calls and stepping through each one would take forever.

Rather, we would like only to suspend execution when it looks like something might be wrong. In this example, we see that the `log` function is producing `NaN`'s. Why don't we just suspend execution when the log function has produced an `NaN`? We can use `trace` to do exactly this.

In the `nLL` function, there is the line

```
lz <- log(z)
```

What we want to do is insert some code after this line which essentially implements the following logic: "If `lz` contains an `NaN`'s, suspend execution and let me browse the environment to see what went wrong." We can do this with the following call to `trace`:

```
> trace("nLL", quote(if(any(is.nan(lz))) { browser() }), at=4, print=F)
```

There are many arguments that need explanation here. The first argument to `trace` is the name of a function. This can be a quoted or a non-quoted string – here we have used a quoted string. The second argument is the code you want to insert. This can either be the name of a function or it can be an unevaulated expression. In this example we have chosen to use an unevaluated expression. The expression we have inserted into `nLL` is the following conditional statement:

```
if(any(is.nan(lz))) {
    browser()
}
```

The code we've decided to insert simply invokes the `browser` function (see Section 3.2.1) if any elements of `lz` have the value `NaN`. This expression has to be put into the `quote` function so that R does not evaluate the code, rather it simply inserts the statements in to the `nLL` function. Without the `quote` function, R would try to evaluate the `if` statement within the `trace` function and it wouldn't make any sense.

The `at` argument tells `trace` were to insert the new code. Here we've instructed `trace` to insert the code before the fourth statement. There's no need to worry about counting statements in your function. This can be done with the following trick.

```
> as.list(body(nLL))
[[1]]
{

[[2]]
z <- mu * x

[[3]]
lz <- log(z)

[[4]]
L1 <- sum(lz)
```

```
[[5]]
L2 <- mu/2

[[6]]
LL <- -(L1 - L2)

[[7]]
LL
```

Here we see that we would like to insert the conditional statement into the place were the `[[4]]` is. That way we can test to see if any elements of `lz` are of value `NaN`. Notice also that the first real line of code is actually the *second* statement in the function.

What `trace` does is copy the original function code into a temporary location and replaces the original function with a new function containing the inserted code. Now when we print the code to `nLL` we see that there are two versions stored:

```
> nLL
An object of class "functionWithTrace"
function (mu, x)
{
    z <- mu * x
    lz <- log(z)
    {
        if (any(is.nan(lz))) {
            browser()
        }
        L1 <- sum(lz)
    }
    L2 <- mu/2
    LL <- -(L1 - L2)
    LL
}
Slot "original":
function(mu, x) {
    z <- mu * x
    lz <- log(z)
    L1 <- sum(lz)
    L2 <- mu/2
    LL <- -(L1 - L2)
    LL
}
```

The top version is our modified function and the bottom version is the original code. It is the top version that gets executed now.

Let's run the traced version of nLL through the optimizer.

```
> optim(100000, nLL, method = "BFGS", x = pp)
Called from: fn(par, ...)
Browse[1]> where
where 1: fn(par, ...)
where 2: function (par)
fn(par, ...)(-68361335.1446888)
where 3: optim(1e+05, nLL, method = "BFGS", x = pp)

Browse[1]> ls()  ## What objects are in our environment?
[1] "lz" "mu" "x"  "z"
Warning message:
NaNs produced in: log(x)
Browse[1]> str(lz)  ## Give a compact representation of lz
 num [1:94] NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN ...
Browse[1]> mu
[1] -68361335
Browse[1]> Q  ## We found the problem!
```

Now we can see that the problem is the optimizer was using negative values for mu. Since the conditional intensity of a point process can never be negative, negative values of mu are not valid. However, the optimizer does not know this and simply chooses values based on a deterministic search algorithm.

3.4 Browse Function Calls with recover

When using browser, you can only browse the environment in the current function call. You cannot poke around in the environments for *previous* function calls. There may be a situation where you want to suspend execution of a function in one location, but then browse a previous function call to hunt down the bug. In other words, you may want to "jump up" to a higher level in the function call stack.

The recover function can help you in this situation. Let us go back to the three functions f, g, and h defined in Section 3.1. Recall that f calls g, which in turn calls h. The h function has a potential problem because it takes the log of a number then compares the result to another number (in this case 10). If log returns NaN, then h will suffer a fatal error.

The statements in the function body of h can be listed using the trick mentioned in Section 3.3:

```
> as.list(body(h))
[[1]]
{
```

```
[[2]]
r <- log(z)

[[3]]
if (r < 10) r^2 else r^3
```

We want to insert some debugging code in the `[[3]]` location and we do this using `trace`:

```
> trace("h", quote( if(is.nan(r)) { recover() } ), at = 3, print = F)
```

Notice that here we check for `NaN`'s in `r` but instead of invoking `browser`, we use `recover`. The results of this are best shown through an example:

```
> f(23)
[1] -203.1205
> f(-10)

Enter a frame number, or 0 to exit
1:f(-10)
2:g(x)
3:h(y)
Selection: 1  ## Browse the f function
Called from: eval(expr, envir, enclos)
Browse[1]> ls()
[1] "x"
Warning message:
NaNs produced in: log(x)
Browse[1]> x
[1] -10
Browse[1]> c  ## Exit the browser and return to recover menu

Enter a frame number, or 0 to exit
1:f(-10)
2:g(x)
3:h(y)
Selection: 2  ## Browse the g function
Called from: eval(expr, envir, enclos)
Browse[1]> ls()
[1] "y"
Browse[1]> y
[1] -10
Browse[1]> c
```

```
Enter a frame number, or 0 to exit
1:f(-10)
2:g(x)
3:h(y)
Selection: 3  # Browse the h function
Called from: eval(expr, envir, enclos)
Browse[1]> ls()
[1] "r" "z"
Browse[1]> r
[1] NaN
Browse[1]> z
[1] -10
Browse[1]> c

Enter a frame number, or 0 to exit
1:f(-10)
2:g(x)
3:h(y)
Selection: 0  ## Exit the recover function
Error in if (r < 10) r^2 else r^3 : missing value where logical needed
```

There is another common use of `recover` which does not require the use of `trace`. The function `options` controls many global options pertaining to your R session. One of them is the `error` option. The `error` option tells R what to do in the situation where a function must halt execution. By default this is set to `NULL` and R does nothing when a function dies. However, you can set this so that instead of the function quitting when an error occurs, it calls `recover` exactly at the location where the error occurred. Typing in

```
> options(error = recover)
```

will do this for you. Now if we were to call `f(-10)` as in the above example, `recover` would be invoked at exactly the place in the `h` function where the fatal error occurs. There is no need to use `trace` in this case.

4 Final Thoughts

Debugging is typically what programmers do about 90% of the time. This is a sad but not unrealistic fact of life. Given that fact, the creators of R have generously provided useful debugging tools to make programmers' lives a little easier. The debugging tools should be used as much as necessary to minimize the time spent debugging and to maximize the time spent, as John Chambers wrote, "turning ideas into software". However, it is all to easy for a programmer to develop an unhealthy relationship with his/her debugger. This is to be avoided. The debugger should not replace common sense in programming and careful design.