A *noweb* processor for R

Terry Therneau

February 12, 2013

1 Introduction

The `coxme` function for mixed effects analyses based on a Cox proportional hazards model is one of the later additions to the `survival` library in S. It is easily the most complex bit of code in the package from all points of view: mathematical, algorithmic, and S code wise. As such, it seems like a natural candidate for documentation and maintainance using the literal programming model.

> Let us change or traditional attitude to the construction of programs. Instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *humans* what we want the computer to do. (Donald E. Knuth, 1984).

The .Rnw files for the source code of *coxme* are not suitable for processing with Sweave; it's goal is to insert the results of R code execution into a document, mine is to act as the repository and documentation for the code itself. These are completely different and need different tools. Luckily, however, the noweb format of ESS (.Rnw or .Snw files) works perfectly for creating the source files needed for either goal. To process the files I use the *noweb* system of Ramsey. The *notangle* routine extracts R source files from the merged .nw file, and the *noweave* routine creates lovely pdf files via latex.

One problem is that the noweb package will not be found on a typical user machine. Those who use unix can install noweb using the usual apt-get command, but for Windows or MacIntosh it is not so easy. To make the code more self contained, I have written a simple noweb parser in R. The *noweb.sty* file that accompanies my version is much simpler than the one found in the noweb source, because I use the hyperref and fancyvrb packages for all the hard stuff. This file itself is the source for the R noweb parser, with the following files:

```
⟨*⟩=
 ⟨nwread⟩
 ⟨nwparse⟩
 ⟨nwloop⟩
 ⟨nwkillamp⟩
 ⟨notangle⟩
 ⟨noweave⟩
 ⟨tab.to.blank⟩
```

⟨*findverbatim*⟩
⟨*nwletter*⟩

The default for the notangle funciton is to extract the code chunk named '*'.

2 Parsing

The `noweb` package is based on unix pipelines. The first step on all the pipelines is to read in a source file and tag the sections. The R code reads the file into a list, each element of which is labeled as program or text. A noweb file consists of chunks: text chunks are signaled by a line with as the first character followed by a newline or a space, and code chunks by the `<<identifier>>=` string alone on a line, where *identifier* is the name of the code chunk. The first line of the program is assumed to be a text chunk if not preceeded by an ampersand line. We parse this into a simple S object, a list containing `nwtext` and `nwcode` objects. All tabs are turned into blanks to facilitate later processing.

A text object that has words on the same line as the @ is treated differently than one that does not, for breaking lines. Comments on the @ line don't count as a new text line, but were used by earlier versions of noweb to aid in cross-indexing. I do not support this historical use — such comments are tossed away.

The standalone Unix version that I have been using will try to interpret any set of paired angle brackets on one line as the start of a code chunk. If one wants to have such a string left alone, e.g. as an argument to grep, then it can be escaped by using `@<<` as the start of the string. The noweb package follows the Sweave convention that declarations of a code chunk must start at the left margin and be alone on the line, hence this problem never arises. Nevertheless, to be consistent with the noweb documentation it will strip the leading `@` symbol from strings of the form `@<<some text>>`, unless they are in a verbatim environment.

The code first finds that start of all code chunks and text chunks and creates a matrix `temp` whose first row is the starting line of the chunk, and second is the type of chunk: 1=text and 2=code. The endline variable contains the last line of each chunk. It then walks through one chunk at a time. One nuisance is to avoid any apparent code start that is the midst of a verbatim or semiverbatim environment. (My document on how to use noweb encounters this issue.)

For a text chunk, it decides if the very first line was a blank line. If not, then the output is a character vector containing all the lines. If it is, then the blank line is suppressed from the character vector, but a `blankline` attribute is added to remind us that it once was there. This is used to create the Latex output in such a way that the line numbers in the .tex file exactly match those in the .Rnw file, while not introducing extra paragraph breaks. Code chunks have a name extracted and the remainder is passed through the `nwparse` function. The code chunk names are used as names for the components of the `noweb` object.

Finally two checks are run: make sure that any code chunk that is referenced has been defined somewhere, and that there are not any loops.

The nwread function creates an object of class noweb. The final noweb object is a list with objects of class `nwtext` and `nwcode`. The first of these is a character vector of the input lines, with an optional blankline attribute. An `nwcode` object has compontents

- lines: the text of the code

- sourceline: the line number of the start of the text, in the original source file (useful when debugging)
- xindx, xref, indent: a set of vectors with one element for each included chunk that give the relative line number in this code where the inclusion occurs, the name of the included chunk, and the indentation amount for the inclusion.

Noweb code chunks are required to have a name.

The nowebSyntax object is a list that defines the text strings that start chunks along with other options.

⟨*nwread*⟩=
```
nwread <- function(file, syntax) {
    if (!file.exists(file)) stop("input file not found")
    program <- tab.to.blank(readLines(file))
    if (length(program)==0) stop("input file is empty")
    vlines <- findverbatim(program, syntax)
    codestart <- grep(syntax$code, program)
    codestart <- codestart[!(codestart %in% vlines)]
    textstart <- grep(syntax$doc, program)
    program <- nwkillat(program, vlines, syntax)  #get rid of extra @ signs

    # Normally users don't start the program with an @, so assume one
    #  Both will be NULL for a doc with no code at all, hence the "2"
    if (min(codestart, textstart, 2) > 1) textstart <- c(1, textstart)

    temp <- rbind( c(codestart, textstart),
                   c(rep(2, length(codestart)), rep(1, length(textstart))))
    temp <- temp[,order(temp[1,])]
    endline <- c(temp[1,-1] -1, length(program))

    output <- vector("list", ncol(temp))  #number of chunks
    oname <- rep("", ncol(temp))
    for (i in 1:ncol(temp)) {
        if (temp[2,i]==1) { # text
            blankline <- sub("^@ *","", program[temp[1,i]])
            if (blankline=="" || substring(blankline,1,1)=="%") {
                # The line is blank
                if (temp[1,i]==endline[i])
                    text <- vector("character",0)  #Nothing there!
                else text <- program[(temp[1,i]+1):endline[i]]
                attr(text, "blankline") <- TRUE
                }
            else {
                text <- blankline
                if (temp[1,i] < endline[i])
                    text <- c(text, program[(temp[1,i]+1):endline[i]])
                attr(text, "blankline") <- FALSE
                }
            class(text) <- "nwtext"
            output[[i]] <- text
            }

        else {  #code
            cname <-  sub(syntax$code, "\\1", program[temp[1,i]])
            if (temp[1,i] == endline[i]) code <- vector("character", 0)
            else code <- program[(temp[1,i]+1):endline[i]]
            oname[i] <- cname
            output[[i]] <- c(nwparse(code, temp[1,i], syntax))
            }
        }

    names(output) <- oname
    class(output) <- "noweb"
    output
    }
```

The `nwparse` routine looks for references to other code within the lines of a code chunk. These are sequences of the form `<<identifier>>`. The resulting structure has the lines, a list of line numbers that are pointers to other code `xindex`, the name of the other code chunk, and the relative indentation.

⟨*nwparse*⟩=
```
nwparse <- function(lines, sourceline, syntax) {
    # Look for references to other code
    indx <- grep(syntax$coderef, lines)
    if (length(indx)) {
        xref <- sub(syntax$coderef, "\\1", lines[indx])
        indent <- sub("<<.*", "", lines[indx])
        out <- list(lines=lines, xref=xref, indent=indent, xindex=indx)
        }
    else out <- list(lines=lines, xref=NULL)

    out$sourceline <- sourceline #original line number in the source file
    class(out) <- "nwcode"
    out
    }
```

The code will fail when expanding a structure with a closed loop of references. This finds such loops. The return value gives the shortest loop found. I only report one because the same loop will appear multiple times, once for each starting point that can enter it.

⟨*nwloop*⟩=
```
nwloop <- function(code) {
    xref <- lapply(code, function(x)
                   if (class(x)=="nwcode") unique(x$xref) else NULL)

    nwchase <- function(chain) {
        xtemp <- xref[[chain[1]]]  #routines called by the head of the chain
        if (length(xtemp) ==0) return(NULL)

        for (i in 1:length(xtemp)) {
            if (!is.na(match(xtemp[i], chain))) return(c(rev(chain), xtemp[i]))
            temp <- nwchase(c(xtemp[i], chain))
            if (!is.null(temp)) return(temp)
            }
        NULL
        }

    cnames <- names(code)
    temp <- lapply(cnames[cnames!=""], nwchase)
    templen <- sapply(temp,length)
    if (any(templen) > 0)
        temp[[min(which(templen==min(templen[templen>0])))]]
    else NULL
    }
```

The extra at sign rule: if we see the the definition of a code chunk prefaced by `@`, and those characters are not inside a `[[]]` or a verbatim clause, pair, then remove the @ sign. This is part of the original noweb specification and I'm playing nice with that spec: if you wanted to ensure that a pair of angle brackets would not trigger "I see the start of a code chunk" then it could be preceded by an ampersand, which would then be stripped off in the final output.

```
⟨nwkillamp⟩=
 nwkillat <- function(program, vlines, syntax) {
     suspectlines <- grep(syntax$escapeat, program)
     suspectlines <- suspectlines[!(suspectlines %in% vlines)]

     # This is slower than Hades, but there are nearly always 0 lines in the
     #  the suspectlines set, and rarely more than 3
     for (i in suspectlines) {
         line <- strsplit(program[i], split='') #make it into a character vector
         inplay <- 1:length(line)  #index to characters not yet exempted
         while(TRUE) {
             temp <- paste(line[inplay], collapse='')
             rtemp <- regexpr(syntax$verb, temp)
             if (rtemp >0) {
                 vchar <- (line[inplay])[rtemp+5]
                 end <- min(0, which(line[inplay[-(1:(rtemp+5))]] == vchar))
                 inplay <- inplay[-(rtemp:(rtemp+5+end))]
                 }
```

```
            else if ((rtemp <- regexpr(syntax$sqexpr, temp)) >0) {
                inplay <- inplay[-(rtemp:(rtemp+attr(rtemp, 'match.length')))]
                }
            else break
            }
        # Remove the @ signs
        keep <- rep(TRUE, length(temp))
        while(1) {
            rtemp <- regexpr(syntax$escapeat, paste(line[inplay], collapse=''))
            if (rtemp>1) {
                line[inplay][rtemp] <- ' '
                keep[inplay[rtemp]] <- FALSE
                }
            else break
            }
        if (any(!keep)) program[i] <- paste(line[keep], collapse='')
    }
    program
}
```

Find any lines that are part of a verbatim environment. An assumption here is that there is nothing important (a code chunk) preceding but on the same line as the `begin{verbatim}`, nor anything of that type after the end of the environment. We first have to get rid of such phrases that live inside a`\verb` clause though, or this paragraph itself would trip us up. This code does not have to be all that good, since it's only purpose is to flag lines that the nwread routine should ignore when looking for the start of code chunks, or nwkillat.

⟨*findverbatim*⟩=
```
findverbatim <- function(code, syntax){
    #Now find paired begin/end clauses
    lines <- NULL

    vstart <- paste("^\\\\begin\\{", syntax$verbatim, "\\}", sep='')
    vend <- paste("\\\\end\\{", syntax$verbatim, "\\}", sep='')
    for (i in 1:length(vstart)) {
        start <- grep(vstart[i], code)
        end   <- grep(vend[i], code)
        if (length(start) != length(end))
            stop(paste("Mismatched", syntax$verbatim[i], "pair"))
        lines <- c(lines, unlist(apply(cbind(start, end), 1,
                                       function(x) x[1]:x[2])))
    }
    sort(unique(lines))
}
```

3 Notangle

The primary reason for wanting an R version of noweb is the notangle function, which extracts and writes out a named R file from the noweb source. This allows the `coxme` package to be built on machines that do not have the standalone noweb package installed. The primary work is the recursion that occurs when one code fragment references another, and maintaining the relative indentation of the point at which it is called.

If no target is given, the default is to extract the target named '*' if it exists, in keeping with the standalone noweb code. If there is no such target I extract the first code chunk.

```
⟨notangle⟩=
 notangle <- function(file, target='*', out, syntax=nowebSyntax, ...) {
     if (inherits(file, "noweb")) input <- file
     else {
         if (.Platform$OS.type == "windows")
             file <- chartr("\\", "/", file)
         input <- nwread(file, syntax)
     }

     if (missing(out)) {
         if (target=='*') {
             # Use the file name
             out <- paste(sub("\\.[^\\.]*$", "", basename(file)), "R", sep='.')
             }
         else out <- paste(target, "R", sep='.')
         }

     cname <- names(input)
     indx <- match(target, cname)
     if (is.na(indx)) {
         if (missing(target) && any(cname != ''))
             target <- (cname[cname!=''])[1]
         else stop(paste("Code chunk", target, "not found in source file"))
         }

     # Verify that there are no loops
     temp <- nwloop(input)
     if (length(temp))
         stop(paste("Code structure has circular references: ",
                    paste(temp, collapse=" --> ")))

     program <- nwextract(input, target, prefix="")

     if (length(out)) cat(program, file=out, sep='\n')
     invisible(program)
     }
```

Here is the actual workhorse function. It extracts a named code chunk, recursively inserting other named ones when they are referenced. If there are inclusions the data is first broken up into a list: element 1 is the start to the first inclusion, element 2 contains the result of a call to nwextract on the first inclusion, element 3 from first inclusion to second inclusion, etc. Some of the odd elements may be empty, for instance if two inclusion lines abut. The ifelse near the end preserves blank lines, no indentation prefix is added to them. It is there mostly to make output exactly match that of the original notangle program.

Note that there can be multiple code chunks with the same name: it is standard in noweb to display a chunk bit by bit as we comment on its structure. Hence the first line below can't be `code[target]` as that would fetch only the first piece. The for loop replaces each chunk with it's expansion one by one; at the end we unlist the result.

⟨*notangle*⟩=
```
nwextract<- function(code, target, prefix="") {
    mycode <- code[names(code)==target]
    if (length(mycode)==0)
        stop(paste("Program chunk '", target, "' not found", sep=""))

    for (chunk in 1:length(mycode)) {
        ctemp <- mycode[[chunk]]
        if (length(ctemp$xref) ==0) temp <- ctemp$lines
        else {
            inclusions <- length(ctemp$xref)
            temp <- vector("list", 2*inclusions +1)
            for (i in 1:length(ctemp$xref))
                temp[[2*i]] <- nwextract(code, ctemp$xref[i], ctemp$indent[i])
            start <- c(1, ctemp$xindex+1) #start and end of non-inclusions
            end   <- c(ctemp$xindex-1, length(ctemp$lines))
            for (i in 1:length(start))
                if (start[i]<=end[i])
                    temp[[2*i -1]] <- ctemp$lines[start[i]:end[i]]
            temp <- unlist(temp)
            }
        mycode[[chunk]] <- ifelse(temp=="", "", paste(prefix, temp, sep=''))
        }
    as.vector(unlist(mycode))   #kill any names added to the vector
    }
```

Convert tabs to blanks. If a tab occurs at position 1 then 8 blanks get added, if at position 2 then 7 blanks get added, etc. If there are two tabs in a row, then the second one is at postion 9, not 2 – you have to do them sequentially. The key is to restart your count at 1 after each tab. The blanks variable contains various size inserts.

⟨*tab.to.blank*⟩=
```
tab.to.blank <- function(x, tabstop=8) {
    blanks <- rep(" ", tabstop)
    for (i in (tabstop-1):1) blanks[i] <- paste(blanks[i +0:1], collapse='')
    temp <- strsplit(x, '')
    linefix <- function(x) {
        n <- length(x)
        if (n==0) ""
        else {
            since.last.tab <- 1:n - cummax(ifelse(x=='\t', 1:n, 0))
            newx <- ifelse(x=='\t', blanks[1+ since.last.tab%%tabstop], x)
            paste(newx, collapse='')
            }
        }
    unlist(lapply(temp, linefix))
    }
```

4 Noweave

The noweave processor is more challenging, largley because of the need to add indexing. Like noweb, we take care to make the result of noweave have exactly the same number of lines as the input source code, so that Latex error messages give correct line numbers.

```
⟨noweave⟩=
 noweave <- function(file, out, indent=1, syntax=nowebSyntax, ...) {
    if (class(file)=="noweb") input <- file
    else {
        if (.Platform$OS.type == "windows")
            file <- chartr("\\", "/", file)
        input <- nwread(file, syntax)
    }
    nchunk <- length(input)
    chunktype <- sapply(input, "class")
    ⟨rep-double-brackets⟩

    if (missing(out))
        out <-  paste(sub("\\.[^\\.]*$", "", basename(file)), "tex", sep='.')
    con <- file(out, open="w")

    ⟨weavecode-count⟩
    if (indent>0) ispace <- paste(rep(" ", indent), collapse='')
    cname <- names(input)
    for (i in 1:length(input)) {
        chunk <- input[[i]]
        if (class(chunk)=="nwtext") cat(chunk, sep="\n", file=con)
        else {
            ⟨weavecode⟩
            }
    }
    close(con)
    cat("\n", sprintf("You can now run (pdf)latex on %s",
                      sQuote(out)), "\n", sep= " ")
}
```

When outputting code chunks we hyperlink each to any prior or succeeding occurences of the chunk. An important tool for this is a simple count of the number of occurrences of each label.

⟨*weavecode-count*⟩=
```
temp <- c(names(input), unlist(lapply(input, function(x)
                                   if (is.list(x)) x$xref else NULL)))
ncount <- table(temp[temp != ""])
ncount2 <- 0*ncount  # number so far, same names, but zeroed
```

Because we want to use the hyperref command the defined verbatim environment `nwchunk` has left the backslash, {, and } characters active. We first need to escape these. Then process the chunks one by one, adding the hyperlinks.

⟨*weavecode*⟩=
```
chunk$lines <- gsub("\\", "{\\textbackslash}", chunk$lines, fixed=TRUE)
chunk$lines <- gsub("{", "\\{", chunk$lines, fixed=TRUE)
chunk$lines <- gsub("}", "\\}", chunk$lines, fixed=TRUE)
chunk$lines <- gsub("\\{\\textbackslash\\}", "{\\textbackslash}",
                    chunk$lines, fixed=TRUE)
```

The last line above undoes damage that 2 and 3 did to textbackslash insertions.

⟨*weavecode*⟩=
```
cn <- cname[i]
ncount2[cn] <- ncount2[cn] +1
# The label for the chunk
if (ncount[cn]==1)   # has no references
    cat("\\begin{nwchunk}\n\\nwhypn{", cn, "}=\n",
        sep='', file=con)
else {
    if (ncount2[cn]==1)   #first instance of the name
       cat("\\begin{nwchunk}\n\\nwhypf{", cn, 1, "}{", cn, "}{", cn, 2,
           "}=\n", sep='', file=con)
    else if (ncount2[cn]== ncount[cn])  #last instance of the name
        cat("\\begin{nwchunk}\n\\nwhypb{", cn, ncount[cn], "}{", cn, "}{",  cn,
            ncount[cn]-1, "}=\n", sep='', file=con)
    else #both
        cat("\\begin{nwchunk}\n\\nwhyp{", cn, ncount2[cn], "}{", cn, "}{", cn,
            ncount2[cn]-1, "}{", cn, ncount2[cn]+1,
            "}=\n", sep='', file=con)
}
```

Now replace any references to other chunks with the appropriate reference

⟨*weavecode*⟩=
```
if (!is.null(chunk$xref)) {
    for (rr in 1:length(chunk$xref)) {
        cn <- chunk$xref[rr]
        ncount2[cn] <- ncount2[cn] +1
        if (ncount[cn] ==1) # has no references
            new <- paste("\\\\nwhypn{", cn, "}", sep='')
        else {
            if (ncount2[cn]==1)  #first instance
                new <- paste("\\\\nwhypf{", cn, 1, "}{", cn, "}{", cn, 2, "}",
                             sep='')
            else if (ncount2[cn] == ncount[cn]) #last instance
                new <- paste("\\\\nwhypb{", cn, ncount[cn], "}{", cn, "}{",
                             cn, ncount[cn]-1, "}", sep='')
            else #both
                new <- paste("\\\\nwhyp{", cn, ncount2[cn], "}{", cn, "}{",
                             cn, ncount2[cn]-1, "}{", cn, ncount2[cn]+1, "}",
                             sep='')
        }
        chunk$lines[chunk$xindex[rr]] <- sub(syntax$coderef, paste(chunk$indent[rr],
                                                                   new, sep=''),
                                             chunk$lines[chunk$xindex[rr]])
    }
}

#write it out
if (indent==0) cat(chunk$lines, sep='\n', file=con)
else cat(paste(ispace, chunk$lines, sep=''), sep='\n', file=con)
cat("\\end{nwchunk}\n", file=con)
```

In noweb one can use [[text]] to set "text" in typewriter font. Replace this with texttt{text}. This replacement is not done in code or within any other environment (verbatim, math, tables, figures, ...) The bugaboo for the code is someone who puts other things on the same line as a latex begin or end statement, which means to be careful we need to keep track of partial lines; doing this is 9/10 of the effort.

Given a chunk, a piece of target text, and a 2 element vector containing the starting line and position within the line, the lookahead function finds the first instance of that target. It returns a two element vector containing the first line/position value after the target. If it can't find a match it returns a position of zero.

⟨*rep-double-brackets*⟩=
```
lookahead <- function(chunk, text, start) {
    # Return the first line #, pos# in the input that is after the text.
    # first look at the starting line
   indx <- gregexpr(text, chunk[start[1]], fixed=T)[[1]]
    if (any(indx >= start[2])) {
        indx <- min(indx[indx>= start[2]])
        if (indx + nchar(text) >= nchar(chunk[start[1]])) c(start[1]+1, 1)
        else c(start[1], indx + nchar(text))
        }
    else { #get first match on later lines
        indx <- regexpr(text, chunk, fixed=T)
        temp <- which(indx >0)

       if (any(temp > start[1])) {
            keep <- min(temp[temp > start[1]])
            end <- indx[keep]+ nchar(text)
            if (end > nchar(chunk[keep])) c(keep+1, 1)
            else c(keep, end)
            }
        else c(1+ length(chunk), 0)  #no match found
        }
    }
```

R (at least in my hands) is not really the right language for text processing. But I don't want to depend on outside code, and programs are small so speed is not an issue. There are two holes in the code. First, I don't check for the built in math modes. However, two starting square brackets would be very unusual notation so we expect no problems there. Second I don't deal with nested lists, i.e., you can have a `\begin{itemize}` within an itemize list. Again, I think this is okay. What is the real target are verbatim environments.

Walk down the chunks of the input one by one. If we are doing the very first one, then start by skipping forward to the `\begin{document}` line. Otherwise we start at indx = line 1, position 1. Then

1. Create `lines`, which is the text yet to be looked at.
2. Find the next instance of `\begin{`, `\verb`, or `[[`. Process the object, and then skip ahead.
3. Repeat until the code is done.

```
⟨rep-double-brackets⟩=
 # First chunk is always the prolog
 for (i in 1:length(input)) {
     chunk <- input[[i]]
     if (i==1) {
         indx <- lookahead(chunk, "\\begin{document", c(1,1))
         if (indx[2] ==0) stop("No begin{document} found, I'm confused")
         }
     else indx <- c(1,1)

     while(class(chunk)== "nwtext" && indx[1] <= length(chunk)) {
         # Find the next thing of interest
         # tline is what's left of the current line
         tline <- substring(chunk[indx[1]], indx[2], nchar(chunk[indx[1]]))
```

```
lines <-  c(tline, chunk[-(1:indx[1])], c("\\begin \\verb"))
temp1 <- grep("\\begin{", lines, fixed=TRUE)
temp2 <- grep("\\verb",  lines, fixed=TRUE)
temp3 <- grep("[[",  lines, fixed=TRUE)

if (length(temp3) ==0) break  #no potential replacements
else {
    nextlineno <- min(c(temp1, temp2, temp3))
    nextline <- lines[nextlineno]
    pos1 <- regexpr("\\begin{", nextline, fixed=TRUE)
    pos2 <- regexpr("\\verb", nextline, fixed=TRUE)
    pos3 <- regexpr("[[", nextline, fixed=TRUE)

    if (pos1 >0 && (pos2<0 || pos2> pos1) && (pos3<0 || pos3> pos1)) {
        # the next thing is a begin clause
        target <- sub("}.*", "",
                  substring(nextline, pos1+attr(pos1, "match.length")))
        indx <- lookahead(chunk, paste("\\end{", target, "}", sep=''),
                      c(indx[1] + nextlineno -1, pos1))
    }
    else if (pos2 >0 && (pos3<0 || pos3 > pos2)) {
        # the next thing is a verb clause
        target <- substring(nextline, pos2+5, pos2+5)
        indx <- lookahead(chunk, target, c(indx[1] + nextlineno -1,
                          pos2+6))
    }
    else {
        # found a [[, do the replacement
        origline <- indx[1] + nextlineno -1
        ltemp <- nwletter(chunk[origline])
        if (nextlineno >1 || indx[2] ==1) {
            #replace the whole line
            chunk[origline] <- sub("(\\[\\[)([^]]*)(]])",
                                paste("\\\\Verb", ltemp, "\\2", ltemp, sep=''),
                                  chunk[origline])
            }
        else { #replace the right half of the line
            temp <-sub("(\\[\\[)([^]]*)(]])",
                       paste("\\\\Verb", ltemp, "\\2", ltemp, sep=''),
                       nextline)

            chunk[origline] <- paste(substring(chunk[origline], 1,
                                              indx[2]-1),
                                temp, sep='')
        }
        indx <- c(indx[1], indx[2]+6)
```

```
                }
            }
        } #end of while loop
        input[[i]] <- chunk
}
```

This little function finds a letter that isn't in the given string, one that can be used to delineate a Verb command.

```
⟨nwletter⟩=
 nwletter <- function(x, try=c("!", "?", "*", "+")){
     for (i in 1:length(try)) {
         if (!grepl(try[i], x,fixed=TRUE)) break
         }
     try[i]
     }
```