```
          1,0,0,0,-1,-1,0,0,-1,0,0)
R> C <- matrix(C, ncol=5)
R> cv <- C %*% V %*% t(C)
R> cr <- matrix(rep(0, ncol(cv)^2),
                ncol=ncol(cv))
R> for (i in 1:5) {
     for (j in 1:5) {
       cr[i,j] <- cv[i,j] / sqrt(cv[i,i]*cv[j,j])
     }
   }
R> delta <- rep(0,5)
R> myfct <- function(q, alpha) {
     lower <- rep(-q, ncol(cv))
     upper <- rep(q, ncol(cv))
     pmvt(lower, upper, df, cr, delta,
          abseps=0.0001)$value - alpha
   }
R> round(uniroot(myfct, lower=1, upper=5,
                 alpha=0.95)$root, 3)
[1] 2.561
```

Here `n` is the sample size vector of each level of the factor, `V` is the covariance matrix of β. With the contrasts $\mathbf{C}$ we can compute the correlation matrix `cr` of $\mathbf{C}\beta$. Finally, we are interested in the 95% quantile of W. A wrapper function `myfct` computes the difference of the multivariate t probability for quantile `q` and `alpha`. The `alpha` quantile can now be computed easily using `uniroot`. The 95% quantile of W in this example is 2.561; reference (1) obtained 2.562 using 80.000 simulation runs. The computation needs 8.06 seconds total time on a Pentium III 450 MHz with 256 MB memory.

Using package **mvtnorm**, the efficient computation of multivariate normal or t probabilities is now available in R. We hope that this is helpful to users / programmers who deal with multiple testing problems.

Bibliography

[1] Don Edwards and Jack J. Berry. The efficiency of simulation-based multiple comparisons. *Biometrics*, 43:913–928, December 1987. 28, 29

[2] A. Genz and F. Bretz. Numerical computation of multivariate t-probabilities with application to power calculation of multiple contrasts. *Journal of Statistical Computation and Simulation*, 63:361–378, 1999. 27, 28

[3] Alan Genz. Numerical computation of multivariate normal probabilities. *Journal of Computational and Graphical Statistics*, 1:141–149, 1992. 27, 28

[4] Alan Genz. Comparison of methods for the computation of multivariate normal probabilities. *Computing Science and Statistics*, 25:400–405, 1993. 27

[5] P. D. Watson, M. B. Wolf, and I. S. Beck-Montgemery. Blood and isoproterenol reduce capillary permeability in cat hindlimb. *The American Journal of Physiology*, 252:H47–H53, 1987. 28

Torsten Hothorn
Friedrich-Alexander-Universität Erlangen-Nürnberg
Institut für Medizininformatik, Biometrie und Epidemiologie
Waldstraße 6, D-91054 Erlangen
Torsten.Hothorn@rzmail.uni-erlangen.de

Frank Bretz
Universität Hannover
LG Bioinformatik, FB Gartenbau
Herrenhäuser Str. 2
D-30419 Hannover
bretz@ifgb.uni-hannover.de

Alan Genz
Department of Mathematics
Washington State University
Pullman, WA 99164-3113 USA
alangenz@wsu.edu

The first author gratefully acknowledges support by Deutsche Forschungsgemeinschaft, grant SFB 539 / A4.

Programmer's Niche

Edited by Bill Venables

Save the environment

When you start to learn how to program in S you don't have to get very far into it before you find that the scoping rules can be rather unintuitive. The sort of difficulty that people first encounter is often something like the following (on S-PLUS 2000):

```
> twowaymeans <- function(X, f)
    apply(X, 2, function(x) tapply(x, f, mean))
> twowaymeans(iris[,1:2], iris$Species)
Error in FUN(X): Object "f" not found
Dumped
```

The dismay expressed by disappointed neophytes on S-news is often palpable. This is not helped by the people who point out that on R it does work because of the more natural scoping rules:

```
> twowaymeans(iris[,1:2], iris$Species)
```

```
           Sepal.Length Sepal.Width
setosa            5.006       3.428
versicolor        5.936       2.770
virginica         6.588       2.974
```

Some people even claim to have chosen R over S-PLUS for the sake of the scoping rules alone. Strange, but true.

The different scoping rules are just one consequence of a feature of R, namely that its functions have an "environment" attached to them which is usually called the function closure. In S the search path is the same for all functions, namely the local frame, frame 1, frame 0 and the sequence of data directories or attached objects. This can be altered on the fly and functions may be made to look at, and interfere with, non-standard places on the search path but this is usually regarded as a device strictly for the excessively brave or the excessively lucky. For some, even using frames 0 or 1 at all is considered rather reckless.

This is the territory where R and S diverge very markedly indeed. How is R so different? I will leave readers to follow up this interesting story using the standard references. All I will try to do here is give an extended example that I hope motivates that kind of followup study. In fact I intend to give the "other story" that I referred to so tantalizingly in the first Programmer's Niche article, of course.

Function closures

We can think of the function closure as a little scratchpad attached to the function that initially contains some objects on which its definition may depend. This turns out to be a useful place to put other things which the function needs to have, but which you don't want to re-create every time you call the function itself. If you understand the concept of frame 0 (the frame of the session) in S, this is a bit like a local frame 0, but for that function alone.

Subsets revisited with caching

In my last article on profiling I used an example of a recursive function to generate all possible subsets of size r of a fixed set of size n. The result is an $\binom{n}{r} \times r$ matrix whose rows define the subsets. Some time ago I devised a way of speeding this process up in S by storing (or 'caching') partial results in frame 0. Doug Bates then pointed out that it could be done much more cleanly using function closures in R.

Thinking back on the subset calculation, notice that if you have all possible subsets of size r of the integers $1, 2, \ldots, n$ as a matrix then the subsets of any set of size n stored in a vector `v` can be got by giving `v` the elements of this matrix as an index vector and using the result to fill a similar matrix with the chosen elements of `v`. (Oh well, think about it for a bit.)

The way we generate all possible subsets of size r from n involves repeatedly generating all possible subsets of smaller sizes from a smaller sets. What we are going to do now is generate these indexing vectors and store them in the the function closure. The index vector for subsets of size 4 from sets of size 10, say, will be given the non-standard name, `4 10`. (It is no penalty to use non-standard names here since these objects will always be accessed indirectly.) Then when we need to generate a set of subsets we will check to see if the index vector to do it is cached in the environment first. If it is we do the job by a single index computation; if not we first generate the index by a recursive call and then use it.

Actually it is one of those times when the code is easier to read than an explanation of it. However even the code is not all that easy. The result, however, is a further spectacular increase in speed but at the cost of greatly increasing your memory usage. If you have to do this sort of computation repeatedly, though, the advantages of the cached index vectors in the environment are maintained, of course, so the second time round, even for the large number of subsets, the computation is nearly instantaneous (providing you are not hitting a memory limit and repeatedly swapping, swapping, swapping, ...). So the technique is both interesting and potentially important.

The code

To get a function with an explicit environment (and not just the global environment) we are going to do it in the "old way" by writing a function to generate the function itself. OK, here goes:

```
makeSubsets <- function() {

  putenv <- function(name, value)
    assign(name, value,
           envir = environment(Subsets))

  getenv <- function(name)
    get(name, envir = environment(Subsets))

  thereIsNo <- function(name)
    !exists(name, envir = environment(Subsets))

  function(n, r, v = 1:n) {
    v0 <- vector(mode(v), 0)
    if(r < 0 || r > n) stop("incompatible n, r")
    sub <- function(n, r, v) {
      if(r == 0) v0 else
      if(r == n) v[1:n] else {
        if(r > 1) {
          i1 <- paste(n-1, r-1)
          i2 <- paste(n-1, r)
          if(thereIsNo(i1))
            putenv(i1, sub(n-1, r-1, 1:(n-1)))
          if(thereIsNo(i2))
            putenv(i2, sub(n-1, r,   1:(n-1)))
          m1 <- matrix(v[-1][getenv(i1)],
```

```
                    ncol = r-1)
        m2 <- matrix(v[-1][getenv(i2)],
                    ncol = r)
      } else {
        m1 <- NULL
        m2 <- matrix(v[2:n], ncol = 1)
      }
      rbind(cbind(v[1], m1), m2)
    }
  }
  sub(n, r, v)
 }
}

Subsets <- makeSubsets()
```

The local environment will initially contain the small utility functions `getenv`, `putenv` and `thereIsNo` for doing various things with the local environment. Within the function itself the index matrices are referred to by the constructed character strings `i1` and `i2`.

Here are a few little comparisons on my oldish Sun system:

```
## best from last time
> system.time(x <- subsets2(20, 7))
[1] 22.26  3.96 26.29  0.00  0.00
## first time round
> system.time(X <- Subsets(20, 7))
[1] 4.94 0.25 5.38 0.00 0.00
## second time
> system.time(L <- Subsets(20, 7, letters))
[1] 1.90 0.04 2.00 0.00 0.00
```

These times are actually quite variable and depend a lot on what is going on with the machine itself. Note that with the fastest function we devised last time the computation took about 26 seconds total time. With the caching version the same computation took 5.38 seconds total time the first time and only 2 seconds the next time when we did the same computation with a different set.

Compression

There is a final speculative twist to this story that I can't resist throwing in even though its usefulness will be very machine dependent.

It must be clear that storing oodles of very large index vectors in the local environment will incur a memory overhead that might well become a problem. Can we compress the vectors on the fly in any way to cut down on this? The only way I have been able to see how to do this has been to use so-called "run length encoding". Given a vector, `v`, the function `rle` finds the lengths of each run of consecutive identical values and returns a list of two components: one giving the values that are repeated in each run and the other the lengths of the runs. This is the inverse operation to the one performed by `rep`: if we feed those two components back to `rep` we re-create the original vector.

The function `rle` is one of the neatest examples of slick programming around. It is very short and a nice little puzzle to see how it works. Here is a cut-down version of it that returns a value with list names matching the argument names of `rep`:

```
rle <- function (x) {
  n <- length(x)
  y <- x[-1] != x[-n]
  i <- c(which(y), n)
  list(x = x[i], times = diff(c(0, i)))
}
```

The thing you notice about these subset index vectors (when the subsets are generated in this lexicographic order) is that they do have long runs of repeated values. In fact the run-length encoded version is an object typically only about half as big (in bytes) as the object itself. This produces a compression of the memory requirements, but at an increased computational cost again, of course. To incorporate the idea into our `makeSubsets` function we need to include this specialised version of `rle` in the local environment as well and to modify the getting and putting functions to include the encoding and decoding:

```
putenv <- function(name, value)
  assign(name, rle(value),
         envir = environment(Subsets))

getenv <- function(name)
  do.call("rep", get(name,
          envir = environment(Subsets)))
```

No modification to the function itself is needed, and this is one advantage of using accessor functions to deal with the local environment, of course.

I think this is an interesting idea, but I have to report that for all the systems I have tried it upon the increased computational penalty pretty well eliminates the gains made by caching. *Quel domage!*

Bill Venables
CSIRO Marine Labs, Cleveland, Qld, Australia
Bill.Venables@cmis.csiro.au