```
.First.lib <- function(libname, pkgname){
    if (!interactive()) return()
    Rcmdr <- options()$Rcmdr
    plugins <- Rcmdr$plugins
    if ((!pkgname %in% plugins) && !getRcmdr("autoRestart")) {
        Rcmdr$plugins <- c(plugins, pkgname)
        options(Rcmdr=Rcmdr)
        closeCommander(ask=FALSE, ask.save=TRUE)
        Commander()
        }
    }
```

Figure 6: The `.First.lib` function from the `RcmdrPlugin.TeachingDemos` package.

J. Fox. The R Commander: A basic-statistics graphical user interface to R. *Journal of Statistical Software*, 14(9):1–42, Aug. 2005. ISSN 1548-7660. URL `http://www.jstatsoft.org/counter.php?id=134&url=v14/i09/v14i09.pdf&ct=1`.

R. M. Heiberger and with contributions from Burt Holland. *RcmdrPlugin.HH: Rcmdr support for the HH package*, 2007. R package version 1.1-4.

R Development Core Team. *Writing R Extensions*. 2007.

G. Snow. *TeachingDemos: Demonstrations for teaching and learning*, 2005. R package version 1.5.

B. B. Welch, K. Jones, and J. Hobbs. *Practical Programming in Tcl/Tk, Fourth Edition*. Prentice Hall, Upper Saddle River NJ, 2003.

John Fox
Department of Sociology
McMaster University
Hamilton, Ontario, Canada
`jfox@mcmaster.ca`

Improvements to the Multiple Testing Package multtest

by Sandra L. Taylor, Duncan Temple Lang, and Katherine S. Pollard

Introduction

The R package **multtest** (Dudoit and Ge, 2005) contains multiple testing procedures for analyses of high-dimensional data, such as microarray studies of gene expression. These methods include various marginal p-value adjustment procedures (the `mt.rawp2adjp` function) as well as joint testing procedures. A key component of the joint testing methods is estimation of a null distribution for the vector of test statistics, which is accomplished via permutations (Westfall and Young, 1993; Ge et al., 2003) or the non-parametric bootstrap (Pollard and van der Laan, 2003; Dudoit et al., 2004). Statistical analyses of high-dimensional data often are computationally intensive. Application of resampling-based statistical methods such as bootstrap or permutation methods to these large data sets further increases computational demands. Here we report on improvements incorporated into **multtest** version 1.16.1 available via both *Bioconductor* and *CRAN*. These updates have significantly increased the computational speed of using the bootstrap procedures.

The **multtest** package implements multiple testing procedures with a bootstrap null distribution through the main user function `MTP`. Eight test statistic functions (`meanX`, `diffmeanX`, `FX`, `blockFX`, `twowayFX`, `lmX`, `lmY`, `coxY`) are used to conduct one and two sample t-tests, one and two-way ANOVAs, simple linear regressions and survival analyses, respectively. To generate a bootstrap null distribution, the `MTP` function calls the function `boot.null`. This function then calls `boot.resample` to generate bootstrap samples. Thus, the call stack for the bootstrap is `MTP -> boot.null -> boot.resample`. Finally, the test statistic function is applied to each sample, and `boot.null` returns a matrix of centered and scaled bootstrap test statistics.

We increased the computational speed of generating this bootstrap null distribution through three main modifications:

1. optimizing the R code of the test statistics;

2. implementing two frequently used tests (two sample t-test and F-test) in C; and

3. integrating an option to run the bootstrap in parallel on a computer cluster.

Changes to Test Statistic Functions

Since the test statistic functions are applied to each bootstrap sample, even small increases in the speed of these functions yielded noticeable improvements to the overall bootstrap procedure. Through profiling the R code of the test statistic functions, we identified several ways to increase the speed of these functions. First, the function `ifelse` was commonly used in the test statistic functions. Changing `ifelse` to an `if...else` construct yielded the largest speed improvements. The `ifelse` function is designed to evaluate vectors quickly but is less efficient at evaluating single values. Small gains were achieved with changing `unique` to `unique.default`, `mean` to `mean.default`, and `sort` to `sort.init`. These changes eliminated the need to assess the object class before selecting the appropriate method when each function was called. The functions `lmX` and `lmY` benefitted from using `rowSums` (i.e., `rowSums(is.na(covar))`) rather than using `apply` (i.e., `apply(is.na(covar),1,sum)`).

The greatest improvements were achieved for the one-sample *t*-test (`meanX`) and the regression tests (`lmX` and `lmY`). We conducted a simulation to evaluate the speed improvements. For the one-sample *t*-test, we randomly generated 100 and 1,000 Normal(0,1) variables for sample sizes of 50 and 100. We tested the null hypotheses that the means for each of the 100 or 1,000 variables were 0. For the two-sample *t*-test, we randomly generated 100 and 1,000 Normal(0,1) variables for two groups consisting of 50 and 100 samples each. For the *F*-test, we used three groups of 50 and 100 samples each. We tested the null hypotheses of equal group means for each of the variables. We evaluated the speed of 1,000 and 10,000 iterations when computing the bootstrap null distributions.

We reduced computational times for the one-sample *t*-test by nearly 60% when the sample size was 50 and from 40% to 46% when the sample size was 100. The number of variables tested and the number of bootstrap iterations did not influence the relative improvement of the revised functions. Changes in R code yielded more modest improvements for the two-sample *t*-test and *F*-test. Computational times were reduced by 25% to 28%, and by about 10%, respectively. As with the one-sample *t*-test, the speed improvements were not affected by the number of variables tested or the number of bootstrap iterations. Sample size had a very small effect for the two-sample test, but did not influence computation speed for the *F*-test.

Because two-sample *t*-tests and *F*-tests are some of the most commonly used tests in **multtest** and only modest improvements were achieved through changes in the R code, we implemented these statistics in C to further increase speed. These modifications took advantage of C code for calculating test statistics in permutation tests that was already in the package. While moving computations into C increases the complexity of the code, the availability of a reference implementation in R allowed us to easily test the new code, easing the transition.

We evaluated the speed improvements of our C implementation with the same approach used to evaluate the R code modifications. By executing the two-sample *t*-test in C, computational time was reduced by about one-half ranging from 46% when the sample size was 100 to 52% for a sample size of 50. The results were more dramatic for the *F*-test; time was reduced by 79% to 82% with the C code implementation. Relative improvements did not vary with the number of bootstrap iterations or variables evaluated. Some further optimization could be done by profiling the C code.

Integration of Parallel Processing

Although we increased the computational speed of generating bootstrap null distributions considerably through improvements to the test statistics code, some analyses still require a long time to complete using a single CPU. Many institutions have computer clusters that can greatly increase computational speed through parallel processing. Bootstrapping is a straightforward technique to conduct in parallel, since each resampled data set is independently generated with no communication needed between individual iterations (Tierney et al., 2007). Thus, synchronizing cluster nodes only entails dispatching tasks to each node and combining the results.

Running the bootstrap on a cluster requires the R package **snow** (Tierney et al., 2007). Through functions in this package, the user can create a cluster object using `makeCluster` and then dispatch jobs to nodes of the cluster through several `apply` functions designed for use with a cluster. We integrated use of a cluster for generating a bootstrap null distribution by adding an argument to the main user interface function (`MTP`) called `cluster`. When this argument is 1 (the default value), the bootstrap is executed on a single CPU. To implement the bootstrap in parallel, the user either supplies a cluster object created using the function `makeCluster` in **snow** or identifies the number of nodes to use in a cluster which `MTP` then uses to create a cluster. In this case, the type of interface system to use must be specified in the `type` argument. MPI and PVM interfaces require **Rmpi** and **rpvm** packages, respectively. `MTP` will check if these packages are installed and load them if necessary.

To use a cluster, **multtest** and **Biobase** (required by **multtest**) must be loaded on each node in the cluster. MTP checks if these packages are installed and loads them on each node. However, if these packages are not installed in a directory in R's library search path, the user will need to create a cluster, load the packages on each node and supply the cluster object as the argument to cluster as shown in the following example code. This code loads the **snow** package, makes a cluster consisting of two nodes and loads **Biobase** and **multtest** onto each node of the cluster using clusterEvalQ.

```
library("snow")
cl <- makeCluster(2, "MPI")
clusterEvalQ(cl, {library("Biobase");
   library("multtest")})
```

Use of the cluster for the bootstrap is then accomplished by specifying the cluster object as the argument to cluster in the MTP function.

```
diffMeanData <- matrix(rnorm(100*100),100)
group <- gl(2, 100)
MTP(X=diffMeanData, Y=group,
   test="t.twosamp.unequalvar",
   alternative="two.sided", B=1000,
   method="sd.minP", cluster=cl)
```

In **multtest**, we use the **snow** package function clusterApplyLB to dispatch bootstrap iterations to cluster nodes. This function automatically balances tasks among available nodes. The user can specify the number or percentage of bootstrap samples to dispatch at a time to each node via the dispatch argument. We set the default value for the dispatch argument to 5% based on simulation results. We considered bootstraps with 100, 1,000 and 10,000 iterations and evaluated dispatching between 1 and 2,500 samples at a time with cluster sizes ranging from 2 to 4 nodes. Dispatching small numbers of iterations to each node took the longest (Figure 1); in these cases the I/O time became substantial. Processing times initially declined as the number of iterations transmitted at one time increased but then leveled off at 10 to 25 iterations for 100 iterations, 25 to 50 for 1,000 iterations and 250 to 500 for 10,000 iterations. Based on these results, we chose to dispatch 5% of the bootstrap iterations at a time as the default value.

Overall Speed Improvements

To assess the overall speed improvements achieved, we compared the speed of the original user interface function MTP and supporting test statistic functions with the new interface function and supporting functions. We compared the speed of 10,000, 25,000 and 50,000 bootstrap iterations for one-sample *t*-tests, two-sample *t*-tests and *F*-tests with 100 and 1,000 variables and group sample sizes of 50 and 100. To assess the effect of a cluster, we used a Linux cluster consisting of 3 nodes. Each node was a Dual Core AMD Opteron 2411.127 MHz Processor with 4 GB of memory.

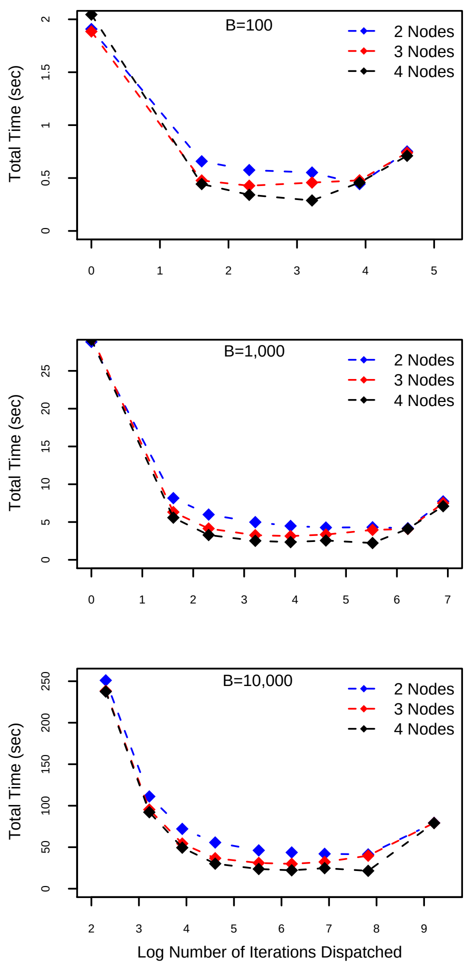

Figure 1: Time to execute 100, 1, 000, and 10, 000 bootstrap iterations for 2, 3, and 4 nodes and varying numbers of samples dispatched at a time.

Computational speed increased substantially with the new methods. For the one-sample *t*-test, computation times were reduced by about 75% for a sample size of 50 and 49% to 69% for a sample size of 100 (Figure 2). Evaluating 1,000 variables in a sample of 50 based on 10,000 bootstrap iterations required 26 minutes with the original functions but only 6.5 minutes with the new functions. The two-sample *t*-test showed the greatest improvement for 50,000 bootstrap iterations, with speed increasing by up to 96%. At 10,000 bootstrap iterations, the new methods were 64% to 90% faster. For 1,000 variables, two groups of 50 and 10,000 bootstrap iterations, computation time was reduced from over an hour to 20 minutes. Speed increases for the *F*-test were more consistent ranging from 78% to 98%. The new methods reduced computation time for 1,000 variables in 3 groups of 50 and based on 10,000 bootstrap iterations from over 3 hours to about 20 minutes. The non-monotone pattern of improvement versus number of bootstrap iterations for the one-sample *t*-test with 100 variables and a sample size of 100 and the *F*-test for 1,000 variables and a sample size of 50 reflects sampling variability of a single observation for each combination.

Summary

Substantial increases in computational speed for implementing the bootstrap null distribution were achieved through optimizing the R code, executing calculations of test statistics in C code, and using a cluster. Through these changes, computational times typically were reduced by more than 75% and up to 96%. Computations that previously required several hours to complete can now be accomplished in half an hour.

Bibliography

S. Dudoit and Y. Ge. Multiple testing procedures, R package, 2005.

S. Dudoit, M. van der Laan, and K. Pollard. Multiple testing. Part I. Single-step procedures for control of general type I error rates. *Statistical Applications in Genetics and Molecular Biology*, 3(1):1–69, 2004. URL `http://www.bepress.com/sagmb/vol3/iss1/art13`.

Y. Ge, S. Dudoit, and T. Speed. Resampling-based multiple testing for microarray data analysis. *TEST*, 12(1):1–44, 2003. URL `http://www.stat.berkeley.edu/users/sandrine/Docs/Papers/Test_spe.pdf`.

K. Pollard and M. van der Laan. Resampling-based multiple testing: Asymptotic control of type I error and applications to gene expression data. *Technical Report 121, Division of Biostatistics, University of California, Berkeley*, pages 1–37, 2003. URL `http://www.bepress.com/ucbbiostat/paper121`.

L. Tierney, A. Rossini, N. Li, and H. Sevcikova. Simple network of workstations, R package, 2007.

P. Westfall and S. Young. *Resampling-based Multiple Testing: Examples and Methods for p-value Adjustment*. John Wiley and Sons, 1993.

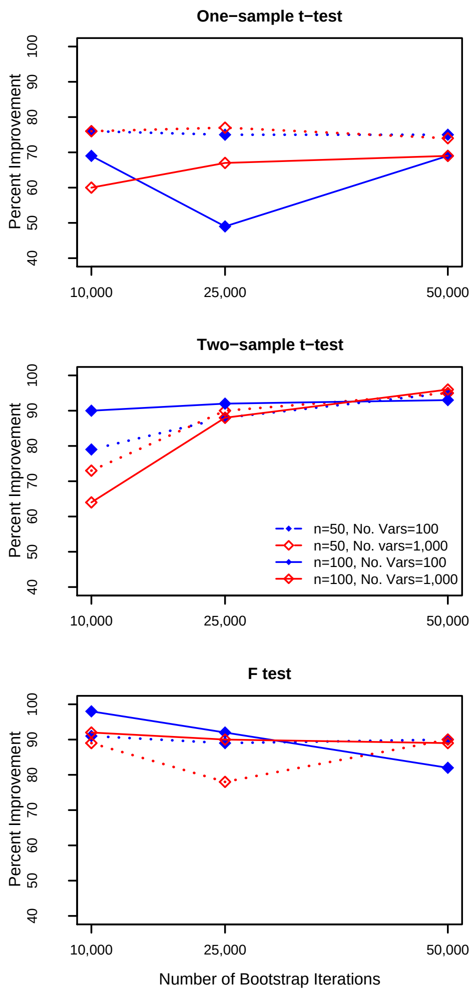

Figure 2: Percent improvement in time to generate bootstrap null distributions based on 10,000, 25,000, and 50,000 iterations for one-sample *t*-tests, two-sample *t*-tests, and F-tests. For the simulations, sample sizes within groups for each test were 50 (dotted lines) or 100 (solid lines). Mean differences were tested for 100 (blue lines) and 1,000 variables (red lines) at each sample size.

Sandra L. Taylor, Duncan Temple Lang, and Katherine S. Pollard
Department of Statistics, University of California, Davis.
`staylor@wald.ucdavis.edu`
`duncan@wald.ucdavis.edu`
`kpollard@wald.ucdavis.edu`