called `stats.`*type*, `sd.`*type* and `limits.`*type*, respectively, for a new chart of type `"`*type*`"`. The following code may be used to define a standardized p chart:

```
stats.p.std <- function(data, sizes)
{
  data <- as.vector(data)
  sizes <- as.vector(sizes)
  pbar <- sum(data)/sum(sizes)
  z <- (data/sizes - pbar)/sqrt(pbar*(1-pbar)/sizes)
  list(statistics = z, center = 0)
}

sd.p.std <- function(data, sizes) return(1)

limits.p.std <- function(center, std.dev, sizes, conf)
{
  if (conf >= 1) { lcl <- -conf
                   ucl <- +conf }
  else
     { if (conf > 0 & conf < 1)
          { nsigmas <- qnorm(1 - (1 - conf)/2)
            lcl <- -nsigmas
            ucl <- +nsigmas }
       else stop("invalid 'conf' argument.") }
  limits <- matrix(c(lcl, ucl), ncol = 2)
  rownames(limits) <- rep("", length = nrow(limits))
  colnames(limits) <- c("LCL", "UCL")
  return(limits)
}
```

Then, we may source the above code and obtain the control charts in Figure 9 as follows:

```
# set unequal sample sizes
> n <- c(rep(50,5), rep(100,5), rep(25, 5))
# generate randomly the number of successes
> x <- rbinom(length(n), n, 0.2)
> par(mfrow=c(1,2))
# plot the control chart with variable limits
> qcc(x, type="p", size=n)
# plot the standardized control chart
> qcc(x, type="p.std", size=n)
```

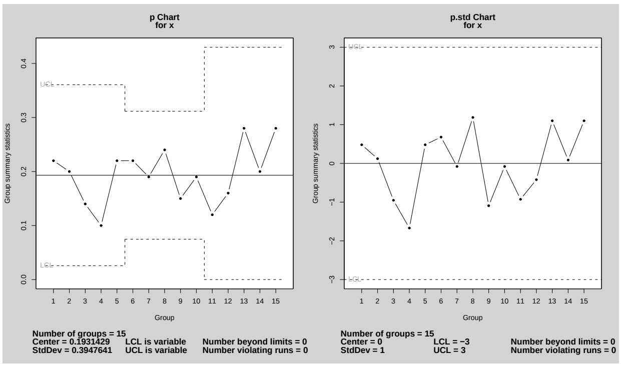

Figure 9: A p chart with non–constant control limits (left panel) and the corresponding standardized control chart (right panel).

Summary

In this paper we briefly describe the **qcc** package. This provides functions to draw basic Shewhart quality control charts for continuous, attribute and count data; corresponding operating characteristic curves are also implemented. Other statistical quality tools available are Cusum and EWMA charts for continuous data, process capability analyses, Pareto charts and cause-and-effect diagram.

References

Montgomery, D.C. (2000) *Introduction to Statistical Quality Control*, 4th ed. New York: John Wiley & Sons.
Wetherill, G.B. and Brown, D.W. (1991) *Statistical Process Control*. New York: Chapman & Hall.

Luca Scrucca
Dipartimento di Scienze Statistiche, Università degli Studi di Perugia, Italy
`luca@stat.unipg.it`

Least Squares Calculations in R

Timing different approaches

by Douglas Bates

Introduction

The S language encourages users to become programmers as they express new ideas and techniques of data analysis in S. Frequently the calculations in these techniques are expressed in terms of matrix operations. The subject of numerical linear algebra - how calculations with matrices can be carried out accurately and efficiently - is quite different from what many of us learn in linear algebra courses or in courses on linear statistical methods.

Numerical linear algebra is primarily based on decompositions of matrices, such as the LU decomposition, the QR decomposition, and the Cholesky decomposition, that are rarely discussed in linear algebra or statistics courses.

In this article we discuss one of the most common operations in statistical computing, calculating least squares estimates. We can write the problem mathematically as

$$\widehat{\beta} = \arg\min_{\beta} \|y - X\beta\|^2 \qquad (1)$$

where X is an $n \times p$ model matrix ($p \leq n$), y is n-dimensional and β is p dimensional. Most statistics texts state that the solution to (1) is

$$\widehat{\beta} = \left(X'X\right)^{-1} X'y \qquad (2)$$

when X has full column rank (i.e. the columns of X are linearly independent).

Indeed (2) is a mathematically correct way of writing a least squares solution and, all too frequently, we see users of R calculating a least squares solution in exactly this way. That is, they use code like `solve(t(X) %*% X) %*% t(X) %*% y`. If the calculation is to be done only a few times on relatively small data sets then this is a reasonable way to do the calculation. However, it is rare that code only gets used only for small data sets or only a few times. By following a few rules we can make this code faster and more stable.

General principles

A few general principles of numerical linear algebra are:

1. Don't invert a matrix when you only need to solve a single system of equations. That is, use `solve(A, b)` instead of `solve(A) %*% b`.
2. Use `crossprod(X)`, not `t(X) %*% X` to calculate $X'X$. Similarly, use `crossprod(X,y)`, not `t(X) %*% y` to calculate $X'y$.
3. If you have a matrix with a special form, consider using a decomposition suited to that form. Because the matrix $X'X$ is symmetric and positive semidefinite, its Cholesky decomposition can be used to solve systems of equations.

Some timings on a large example

For a large, ill-conditioned least squares problem, such as that described in Koenker and Ng (2003), the literal translation of (2) does not perform well.

```
> library(Matrix)
> data(mm, y)
> mmm = as(mm, "matrix")
> dim(mmm)

[1] 1850  712

> sysgc.time(naive.sol <- solve(t(mmm) %*%
+     mmm) %*% t(mmm) %*% y)

[1] 3.64 0.18 4.11 0.00 0.00
```

(The function `sysgc.time` is a modified version of `system.time` that calls `gc()` before initializing the timer, thereby providing more consistent timing results. The data object `mm` is a sparse matrix, as described below, and we must coerce it to the dense matrix `mmm` to perform this calculation.)

According to the principles above, it should be more effective to compute

```
> sysgc.time(cpod.sol <- solve(crossprod(mmm),
+     crossprod(mmm, y)))

[1] 0.63 0.07 0.73 0.00 0.00

> all.equal(naive.sol, cpod.sol)

[1] TRUE
```

Timing results will vary between computers but in most cases the crossprod form of the calculation is at least four times as fast as the naive calculation. In fact, the entire crossprod solution is usually faster than calculating $X'X$ the naive way

```
> sysgc.time(t(mmm) %*% mmm)

[1] 0.83 0.03 0.86 0.00 0.00
```

because the `crossprod` function applied to a single matrix takes advantage of symmetry when calculating the product.

However, the value returned by `crossprod` does not retain the information that the product is symmetric (and positive semidefinite). As a result the solution of (1) is performed using a general linear system solver based on an LU decomposition when it would be faster, and more stable numerically, to use a Cholesky decomposition. The Cholesky decomposition could be used explicitly but the code is awkward

```
> sysgc.time(ch <- chol(crossprod(mmm)))

[1] 0.48 0.02 0.52 0.00 0.00

> sysgc.time(chol.sol <- backsolve(ch,
+     forwardsolve(ch, crossprod(mmm, y),
+      upper = TRUE, trans = TRUE)))

[1] 0.12 0.05 0.19 0.00 0.00

> all.equal(chol.sol, naive.sol)

[1] TRUE
```

Least squares calculations with Matrix classes

The Matrix package uses the S4 class system (Chambers, 1998) to retain information on the structure of matrices returned by intermediate calculations. A general matrix in dense storage, created by the Matrix function, has class "geMatrix". Its crossproduct has class "poMatrix". The solve methods for the "poMatrix" class use the Cholesky decomposition.

```
> mmg = as(mm, "geMatrix")
> class(crossprod(mmg))

[1] "poMatrix"
attr(,"package")
[1] "Matrix"

> sysgc.time(Mat.sol <- solve(crossprod(mmg),
+     crossprod(mmg, y)))

[1] 0.46 0.04 0.50 0.00 0.00

> all.equal(naive.sol, as(Mat.sol, "matrix"))

[1] TRUE
```

Furthermore, any method that calculates a decomposition or factorization stores the resulting factorization with the original object so that it can be reused without recalculation.

```
> xpx = crossprod(mmg)
> xpy = crossprod(mmg, y)
> sysgc.time(solve(xpx, xpy))

[1] 0.08 0.01 0.09 0.00 0.00

> sysgc.time(solve(xpx, xpy))

[1] 0.01 0.00 0.01 0.00 0.00
```

The model matrix mm is sparse; that is, most of the elements of mm are zero. The Matrix package incorporates special methods for sparse matrices, which produce the fastest results of all.

```
> class(mm)

[1] "cscMatrix"
attr(,"package")
[1] "Matrix"

> sysgc.time(sparse.sol <- solve(crossprod(mm),
+     crossprod(mm, y)))

[1] 0.03 0.00 0.04 0.00 0.00

> all.equal(naive.sol, as(sparse.sol, "matrix"))

[1] TRUE
```

The model matrix, mm, has class "cscMatrix" indicating that it is a compressed, sparse, column-oriented matrix. This is the usual representation for sparse matrices in the Matrix package. The class of crossprod(mm) is "sscMatrix" indicating that it is a symmetric, sparse, column-oriented matrix. The solve methods for such matrices first attempts to form a Cholesky decomposition. If this is successful the decomposition is retained as part of the object and can be reused in solving other systems based on this matrix. In this case, the decomposition is so fast that it is difficult to determine the difference in the solution times.

```
> xpx = crossprod(mm)
> class(xpx)

[1] "sscMatrix"
attr(,"package")
[1] "Matrix"

> xpy = crossprod(mm, y)
> sysgc.time(solve(xpx, xpy))

[1] 0.01 0.00 0.01 0.00 0.00

> sysgc.time(solve(xpx, xpy))

[1] 0.01 0.00 0.01 0.00 0.00
```

Epilogue

Another technique for speeding up linear algebra calculations is to use highly optimized libraries of Basic Linear Algebra Subroutines (BLAS) such as Kazushige Goto's BLAS (Goto and van de Geijn, 2002) or Atlas (Whaley et al., 2001). As described in R Development Core Team (2004), on many platforms R can be configured to use these enhanced BLAS libraries.

Most of the fast BLAS implementations focus on the memory access patterns within the calculations. In particular, they structure the calculation so that on-processor "cache" memory is used efficiently in basic operations such as dgemm for matrix-matrix multiplication.

Goto and van de Geijn (2002) describe how they calculate the matrix product $\boldsymbol{AB}$ by retrieving and storing parts of the transpose of $\boldsymbol{A}$. In the product it is the rows of $\boldsymbol{A}$ and the columns of $\boldsymbol{B}$ that will be accessed sequentially. In the Fortran column-major storage convention for matrices, which is what is used in R, elements in the same column are in adjacent storage locations. Thus the memory access patterns in the calculation $\boldsymbol{A}'\boldsymbol{B}$ are generally faster than those in $\boldsymbol{AB}$.

Notice that this means that calculating $\boldsymbol{A}'\boldsymbol{B}$ in R as t(A) %*% B instead of crossprod(A, B) causes A to be transposed twice; once in the calculation of t(A) and a second time in the inner loop of the matrix product. The `crossprod` function does not do any transposition of matrices – yet another reason to use `crossprod`.

Bibliography

J. M. Chambers. *Programming with Data*. Springer, New York, 1998. ISBN 0-387-98503-4. 19

K. Goto and R. van de Geijn. On reducing TLB misses in matrix multiplication. Technical Report TR02-55, Department of Computer Sciences, U. of Texas at Austin, 2002. 19

R. Koenker and P. Ng. SparseM: A sparse matrix package for R. *J. of Statistical Software*, 8(6), 2003. 18

R Development Core Team. *R Installation and Administration*. R Foundation for Statistical Computing, Vienna, 2004. ISBN 3-900051-02-X. 19

R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`). 19

D.M. Bates
U. of Wisconsin-Madison
`bates@wisc.edu`

Tools for interactively exploring R packages

by Jianhua Zhang and Robert Gentleman

Introduction

An R package has the required and perhaps additional subdirectories containing files and information that may be of interest to users. Although these files can always be explored through command line operations both within and outside R, interactive tools that allow users to view and or operate on these files would be appealing to many users. Here we report on some widgets of this form that have been developed using the package *tcltk*. The functions are called `pExplorer`, `eExplorer`, and `vExplorer` and they are intended for exploring packages, examples and vignettes, respectively. These tools were developed as part of the Bioconductor project and are available in the package *tkWidgets* from `www.bioconductor.org`. Automatic downloading and package maintenance is also available using the *reposTools* mechanisms, also from Bioconductor.

Description

Our current implementation is in the form of `tcl/tk` widgets and we are currently adding similar functionality to the *RGtk* package, also available from the Bioconductor website. Users must have a functioning implementation (and be configured for) `tcl/tk`. Once both *tkWidgets* and *widgetTools* have been loaded into the working session the user has full access to these interactive tools.

pExplorer

`pExplorer` allows users to explore the contents of an R package. The widget can be invoked by providing a package name, a path to the directory containing the package, and the names of files and or subdirectory to be excluded from showing by the widget. The default behavior is to open the first package in the library of locally installed R (`.libPaths()`) with subdirectories/files named "Meta" and "latex" excluded if nothing is provided by a user.

Figure 1 shows the widget invoked by typing `pExplorer("base")`.

The following tasks can be performed through the interface:

- Typing in a valid path to a local directory containing R packages and then press the Enter key will update the name of the package being explored and show the contents of the first R package in the path in the `Contents` list box.
- Clicking the dropdown button for package path and selecting a path from the resulting dropdown list by double clicking will achieve the same results as typing in a new path.
- Clicking the `Browse` button allows users to selecting a directory containing R packages to achieve the same results as typing in a new path.
- Clicking the `Select Package` button allows users to pick a new package to explore and the name and contents of the package being explored are updated.