SparseM: A Sparse Matrix Package for R *

Roger Koenker and Pin Ng

November 5, 2002

Abstract

SparseM provides some basic R functionality for linear algebra with sparse matrices. Use of the package is illustrated by a family of linear model fitting functions that implement least squares methods for problems with sparse design matrices. Significant performance improvements in memory utilization and computational speed are possible for applications involving large sparse matrices.

1 Introduction

Many applications in statistics involve large sparse matrices, matrices with a high proportion of zero entries. A typical example from parametric linear regression involves longitudinal data with fixed effects: many indicator variables consisting of a few ones and a large number of zero elements. In nonparametric regression, e.g. smoothing splines design matrices are extremely sparse often with less than 1% of nonzero entries. Conventional algorithms for linear algebra in such situations entail exorbitant storage requirements and many wasteful floating point operations involving zero entries. For some specially structured problems, e.g. banded matrices, special algorithms are available. But recent developments in sparse linear algebra have produced efficient methods for handling unstructured sparsity in a remarkably efficient way.

Exploiting these developments, the package SparseM provides some basic linear algebra functionality for sparse matrices stored in several standard formats. The package attempts to make the use of these methods as transparent as possible by adhering to the method-dispatch conventions of \texttt{R}.\footnote{To that end, we have had to make several familiar functions generic with respect to this package. These include \texttt{\%\%}, \texttt{diag}, \texttt{chol}, \texttt{backsolve}. Our thanks to Brian Ripley for advice on this aspect of the package.} Functions are provided for: coercion, basic unary and binary operations on matrices and linear equation solving.

Our implementation is based on Sparskit (Saad (1994)), which provides one of the more complete collection of subroutines for BLAS like functions and sparse

\footnote{This package should be considered experimental. The authors would welcome comments about any aspect of the package. This document is an R vignette prepared with the aid of \texttt{Sweave}, Leisch(2002). Support from NSF SES 99-11184 is gratefully acknowledged.}
matrix utilities available in the public domain. Our Cholesky factorization and
backsolve routines are based on Ng and Peyton (1993), which still appears to
represent the state of the art for solving linear systems involving symmetric
positive definite matrices.

In Section 2 we discuss in more detail the components of the package, provide
some examples on their use and explain the basic design philosophy. Section 3
contains some performance results while Section 4 discusses some refinements
proposed for future implementations.

SparseM can be obtained from the Comprehensive R Archive Network, CRAN,
at http://cran.r-project.org/.

2 Design Philosophy

In this section we briefly describe some aspects of our design philosophy begin-
ing with the question of storage modes.

2.1 Storage Modes

There are currently more than twenty different storage formats used for sparse
matrices. Each of these formats is designed to exploit particular features of
the matrices that arise in various applications areas to gain efficiency in both
memory utilization and computation. Duff, Erisman and Reid (1986) and Saad
(1994) provide detailed accounts of the various storage schemes. Following
Saad (1994) we have chosen compressed sparse row (csr) format as the primary
storage mode for SparseM. An n by m matrix A with real elements a_{ij}, stored
in csr format consists of three arrays:

\begin{itemize}
  \item ra: a real array of nnz elements containing the non-zero elements of A,
  stored in row order. Thus, if i < j, all elements of row i precede elements
  from row j. The order of elements within the rows is immaterial.
  \item ja: an integer array of nnz elements containing the column indices of the
  elements stored in ra.
  \item ia: an integer array of n + 1 elements containing pointers to the beginning
  of each row in the arrays ra and ja. Thus ia[i] indicates the position in
  the arrays ra and ja where the ith row begins. The last (n + 1)st element
  of ia indicates where the n + 1 row would start, if it existed.
\end{itemize}

The following commands illustrate typical coercion operations.

\footnote{Other sparse storage formats supported in SparseM include compressed sparse column (csc), symmetric sparse row (ssr) and symmetric sparse column (ssc). The data structure of csc format is the same as that of csr format except the information is stored column-wise. The ssr and ssc formats are special cases of csr and csc, respectively, for symmetric matrices, only the information in the lower triangle is stored. We have created new class objects, matrix.csr, matrix.csc, matrix.ssr, matrix.ssc, for each of these four formats.}
> library(SparseM)
[1] "SparseM library loaded"

Attaching package ‘SparseM’:

The following object(s) are masked from package:base :

%*% backsolve chol diag diag<- 

> a <- rnorm(5 * 4)
> a[abs(a) < 0.7] <- 0
> A <- matrix(a, 5, 4)
> A 

[1,] 0.000000 1.3272533 1.0632486 1.5023623
[2,] 0.000000 0.0000000 0.8194708 0.0000000
[3,] 1.5233480 0.0000000 1.2151604 -1.3292917
[4,] -1.0562860 1.4759791 0.9830683 0.0000000
[5,] 0.0000000 0.0000000 0.0000000 0.0000000

> A.csr <- as.matrix.csr(A)
> A.csr 

$ra
 [1] 1.3272533 1.0632486 1.5023623 0.8194708 1.5233484 1.2151604 -1.3292917 -1.0562858 1.4759791 0.9830683

$ja
 [1] 2 3 4 3 1 3 4 1 2 3

$ia
 [1] 1 4 5 8 11 11

$dim
[1] 5 4

attr(,"class")
[1] "matrix.csr"

> as.matrix(A.csr)

[1,] 0.000000 1.327253 1.0632486 1.502362
[2,] 0.000000 0.000000 0.8194708 0.000000

attr(,"class")
[1] "matrix.csr"

> as.matrix(A.csr)
To facilitate testing we have included `read.matrix.hb` and `write.matrix.hb` to deal with matrices in the Harwell-Boeing storage format. A list of sites with extensive collections of sparse matrices in this format can be found at [http://math.nist.gov/MatrixMarket/](http://math.nist.gov/MatrixMarket/). Details on the Harwell-Boeing format can be found in the help files for `read.matrix.hb` and `write.matrix.hb` as well as in the User's Guide for Harwell-Boeing Sparse Matrix Collection at [ftp://ftp.cerfacs.fr/pub/harwell_boeing/](ftp://ftp.cerfacs.fr/pub/harwell_boeing/).

### 2.2 Visualization

The `image` function allows users to explore the structure of the sparsity in matrices stored in `csr` format. In the next example we illustrate the design matrix for a bivariate spline smoothing problem illustrated in Koenker and Mizera (2002). The upper 100 rows of the matrix are an identity matrix, the lower 275 rows represent the penalty component of the design matrix. In this example $X$ has 1200 nonzero entries, roughly 3.2 percent of the number of floating point numbers needed to represent the matrix in dense form. The $X'X$ form of the matrix has 1162 nonzero elements or 11.62 percent of the entries in the full matrix.

\[
\begin{bmatrix}
1.523348 & 0.000000 & 1.215160 & -1.329292 \\
-1.056286 & 1.475979 & 0.983068 & 0.000000 \\
0.000000 & 0.000000 & 0.000000 & 0.000000 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
20 & 40 & 60 & 80 \\
300 & 200 & 100 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
20 & 40 & 60 & 80 \\
100 & 80 & 60 & 40 & 20 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
X \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
X'X \\
\end{bmatrix}
\]
2.3 Indexing and Binding

Indexing and the functions \texttt{cbind} and \texttt{rbind} for the \texttt{matrix.csr} class work just like they do on dense matrices. Objects returned by \texttt{cbind} and \texttt{rbind} operating on objects of the \texttt{matrix.csr} class retain their \texttt{matrix.csr} class attribute.

2.4 Linear Algebra

SparseM provides a reasonably complete set of commonly used linear algebra operations for the \texttt{matrix.csr} class. The general design philosophy for this set of functions is that operations on \texttt{matrix.csr} class will yield an object also in \texttt{matrix.csr} class with a few exceptions mentioned below.

The functions \texttt{t}, and \texttt{\%\%\%} for transposition, and multiplication of \texttt{csr} matrices work just like their dense matrix counterparts and the returned objects retain their \texttt{matrix.csr} class. The \texttt{diag} and \texttt{diag<-} functions for extracting and assigning the diagonal elements of \texttt{csr} matrices also work like their dense matrix counterparts except that the returned objects from \texttt{diag} are dense vectors with appropriate zeros reintroduced. The unary and binary functions in the group generic functions \texttt{Ops} return objects of \texttt{matrix.csr} class.

2.5 Linear Equation Solving

Research on solutions to sparse systems of linear equations has focused primarily on methods based on the Cholesky factorization, and we have followed this approach. There are three functions \texttt{chol}, \texttt{backsolve} and \texttt{solve} to handle a symmetric positive definite system of linear equations. \texttt{chol} performs Cholesky factorization using the block sparse Cholesky algorithms of Ng and Peyton (1993). The result can then be passed on to \texttt{backsolve} with a right-hand-side to obtain the solutions. For system of linear equations that only vary on the right-hand-side, the result from \texttt{chol} can be reused and, saving considerable computing time. The function \texttt{solve}, which combines the use of \texttt{chol} and \texttt{backsolve}, will compute the inverse of a matrix by default, if the right-hand-side is missing.

2.6 Least Squares Problems

To illustrate the functionality of the package we include an application to least squares regression. The group of functions \texttt{slm}, \texttt{slm.fit}, \texttt{slm.fit.csr}, \texttt{summary.slm} and \texttt{print.summary.slm} provide analogues of the familiar \texttt{lm} family. In the current implementation \texttt{slm} processes a formula object in essentially the same way as \texttt{lm}, and calls an intermediate function \texttt{slm.fit}, which in turn calls \texttt{slm.fit.csr} where the actual fitting occurs. Rather than the usual QR decomposition, \texttt{slm.fit.csr} proceeds by backsolving the triangular system resulting from a Cholesky decomposition of the $X'X$ matrix. The sparsity of the resulting structure is usually well preserved by this strategy. The use of sparse methods is quite transparent in the present \texttt{slm} implementation and \texttt{summary.slm} with the
associated print.summary.slm should produce identical output to their cousins in the \texttt{lm} family. However, the speed and memory utilization can be quite dramatically improved. In the following problem, which involves a design matrix that is 1850 by 712 there is a nearly three hundred fold improvement in speed (on a Sun Ultra 2) when we compare \texttt{lm.fit} and \texttt{slm.fit}. The comparison is somewhat less compelling between \texttt{lm} and \texttt{slm} since there is a substantial common fixed cost to the setup of the problems. In addition to the computational time saved there is also a significant reduction in the memory required for large sparse problems. In extreme cases memory becomes a binding constraint on the feasibility of large problems and sparse storage is critical in expanding the range of problem sizes. This is particularly true of applications in smoothing and related image processing contexts.

\begin{verbatim}
> hb.o <- read.matrix.hb(system.file("HBdata", "lsq.rra", package = "SparseM"))
> X <- model.matrix.hb(hb.o)
> y <- model.response.hb(hb.o)
> X1 <- as.matrix(X)
> slm.time <- unix.time(slm.o <- slm(y ~ X1 - 1))
> lm.time <- unix.time(lm.o <- lm(y ~ X1 - 1))
> slm.fit.time <- unix.time(slm.fit(X, y))
> lm.fit.time <- unix.time(lm.fit(X1, y))
> cat("slm time =", slm.time, "\n")
slm time = 9 0.35 14.4 0 0
> cat("lm time =", lm.time, "\n")
lm time = 34.16 0.04 49.62 0 0
> cat("slm.fit time =", slm.fit.time, "\n")
slm.fit time = 0.13 0 0.17 0 0
> cat("lm.fit time =", lm.fit.time, "\n")
lm.fit time = 28.17 0.01 41.15 0 0
> cat("slm Results: Reported Coefficients Truncated to 5 ", "\n")
slm Results: Reported Coefficients Truncated to 5
> sum.slm <- summary(slm.o)
> sum.slm$coef <- sum.slm$coef[1:5,]
> sum.slm
Call:
  slm(formula = y ~ X1 - 1)

\end{verbatim}
Residuals:
          Min       1Q Median       3Q      Max
-1.952e-01 -1.400e-02  5.329e-15  1.442e-02  1.783e-01

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
[1,]   823.3613   0.1274   6460.4  <2e-16 ***
[2,]   340.1156   0.1711   1987.3  <2e-16 ***
[3,]   472.9760   0.1379   3429.6  <2e-16 ***
[4,]   349.3175   0.1743   2004.0  <2e-16 ***
[5,]   187.5595   0.2100    893.3  <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.03789 on 1138 degrees of freedom
Multiple R-Squared: 1,  Adjusted R-squared: 1
F-statistic: 4.504e+07 on 712 and 1138 DF,  p-value: < 2.2e-16

> cat("lm Results: Reported Coefficients Truncated to 5 ", ", 
"

lm Results: Reported Coefficients Truncated to 5

> sum.lm <- summary(lm.o)
> sum.lm$coef <- sum.lm$coef[1:5, ]
> sum.lm

Call:
lm(formula = y ~ X - 1)

Residuals:
          Min       1Q Median       3Q      Max
-0.19522  -0.01400  0.00000  0.01442  0.17833

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
X11  823.3613   0.1274   6460.4  <2e-16 ***
X12  340.1156   0.1711   1987.3  <2e-16 ***
X13  472.9760   0.1379   3429.6  <2e-16 ***
X14  349.3175   0.1743   2004.0  <2e-16 ***
X15  187.5595   0.2100    893.3  <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.03789 on 1138 degrees of freedom
Multiple R-Squared: 1,  Adjusted R-squared: 1
F-statistic: 4.504e+07 on 712 and 1138 DF,  p-value: < 2.2e-16
3 Some Potential Refinements

There are still many features that could be usefully added to the package. Among these we would especially like to see in the near future: crossprod, row, col, eigen, det, svd and the set of functions for QR decomposition. Support for other storage formats might be eventually useful, although csr, csc, ssr, ssc formats seem quite sufficient for most purposes. A major improvement in the slm implementation would be to replace the line

\[ X \leftarrow \text{as.matrix.csr}(\text{model.matrix}(\text{Terms}, m, \text{contrasts})) \]

which coerces the dense form of the regression design matrix produced by model.matrix into the sparse form. Ideally, this would be done with a special .csr form of model.matrix, thus obviating the need to construct the dense form of the matrix. We have not looked carefully at the question of implementing this suggestion, but we hope that someone else might be inspired to do so.

Our primary motivation for \( R \) sparse linear algebra comes from our experience, see e.g. Koenker, Ng and Portnoy (1994) and He and Ng (1999), with interior point algorithms for quantile regression smoothing problems. We hope to report on this experience elsewhere.

References


