The Matrix Package: Programming with S4 Classes

Martin Maechler
ETH Zurich
Switzerland
maechler@R-project.org

R/Rmetrics 2008, Meielisalp
July 1, 2008

Outline
- Introduction
- Introduction to Sparse Matrices in package Matrix
- Matrix: S4 classes and methods
  - Matrix: Goals
  - 3D space of Matrix classes
- Sparse Least Squares

Introduction

Some reasons to perform object oriented programming:
1. productivity increase
2. easier to maintain code
3. reusable code
4. the design tends to follow the objects being modeled
5. encapsulate the representation of objects
6. specialize the behavior of functions to objects

Sparse Matrices in R package Matrix

- The R Package Matrix contains dozens (well, 70) of matrix classes and hundreds\(^1\) of method definitions.
- Has sub-hierarchies of denseMatrix and sparseMatrix.
- This talk: Very basic introduction in some its sparse matrices:

\(^1\)\geq 1234, as of June 2006
The most obvious way to store a sparse matrix is the so called "Triplet" form; (virtual class TsparseMatrix in Matrix):

```r
> A <- spMatrix(10,20, i = c(1,3:8),
+ j = c(2,9,6:10),
+ x = 7 * (1:7))
> A # a "dgTMatrix"
10 x 20 sparse Matrix of class "dgTMatrix"
[1,] . 7 . . . . . . . . . . . . . . . . . .
[2,] . . . . . . . . . . . . . . . . . . . .
[3,] . . . . . . . . 14 . . . . . . . . . .
[4,] . . . . . 21 . . . . . . . . . . . . .
[5,] . . . . . . 28 . . . . 90 . . . . . . .
[6,] . . . . . . . 35 . . . 90 . . . . . . .
[7,] . . . . . . . . 42 . . 90 . . . . . . .
[8,] . . . . . . . . . 49 . . . . . . . . .
[9,] . . . . . . . . . . . . . . . . . . . .
[10,] . . . . . . . . . . . . . . . . . . . .
```

```r
> str(A) # note that *internally* 0-based indices (i,j) are used
Formal class 'dgTMatrix' [package "Matrix"] with 6 slots
..@ i : int [1:7] 0 2 3 4 5 6 7
..@ j : int [1:7] 1 8 5 6 7 8 9
..@ Dim : int [1:2] 10 20
..@ Dimnames:List of 2
.. ..$ : NULL
.. ..$ : NULL
..@ x : num [1:7] 7 14 21 28 35 42 49
..@ factors : list()
```

```r
> A[2:7, 12:20] <- rep(c(0,0,0,(3:1)*30,0), length = 6*9)
> A ## note the added "block" [2:7, 12:20] (90,60,30, many 0)
10 x 20 sparse Matrix of class "dgTMatrix"
[1,] . 7 . . . . . . . . . . . . . . . . . .
[2,] . . . . . . . . . . . . . 30 60 90 . . . .
[3,] . . . . . . . . . . . . . . 30 60 90 . . . .
[4,] . . . . . 21 . . . . . . . . . 30 60 90 . . .
[5,] . . . . . . 28 . . . 90 . . . 30 60 90 . . .
[6,] . . . . . . . 35 . . 90 . . . 30 60 90 . . .
[7,] . . . . . . . . 42 . 90 . . . 30 60 90 . . .
[8,] . . . . . . . . . 49 . . . . . . . . . . . .
[9,] . . . . . . . . . . . . . . . . . . . . . .
[10,] . . . . . . . . . . . . . . . . . . . . . .
```

```r
> A >= 20 # -> logical sparse; nice show() method
10 x 20 sparse Matrix of class "lgTMatrix"
[1,] . . . . . . . . . . . . . . . . . . . .
[2,] . . . . . . . . . . . . . | | | . . . .
[3,] . . . . . . . . . . . . . . | | | . . .
[4,] . . . . . | . . . . . . . . . | | | . .
[5,] . . . . . . | . . . . | . . . . | | | .
[6,] . . . . . . . | . . . | | . . . . | | |
[7,] . . . . . . . . | . . | | | . . . . | |
[8,] . . . . . . . . . | . . . . . . . . . .
[9,] . . . . . . . . . . . . . . . . . . . .
[10,] . . . . . . . . . . . . . . . . . . . .
```

sparse compressed form

Triplet representation: easy for us humbly humans, but can be both made smaller and more efficient for (column-access heavy) operations:
The “column compressed” sparse representation, (virtual class CsparseMatrix in Matrix):

Column compressed ("Csparse")

> Ac <- as(t(A), "CsparseMatrix") # – t(.) : *transposed*  
> str(Ac)

Matrix arithmetic etc – 1 –

> 3 * A # remains sparse

Matrix arithmetic etc – 2 –

Automatic sparse to dense conversion “when needed” :

> A + 1 ## of course is no longer sparse

Matrix products – 1 –

Cross product

> A %*% t(A)

Note that the result must be symmetric, but the %*% operator cannot guess this.  
Rather, use tcrossprod:

> the method dispatch for the * function happens on  
  "signature" <numeric> o <dgTMatrix>.

> The result is dgCMatrix, i.e., "CsparseMatrix", i.e.,  
  compressed sparse.
Matrix products: `crossprod()` & `tcrossprod()`

tcrossprod() was introduced to R only a few versions ago, after being part of the R package `Matrix` for a couple of years:

Def. `crossprod()` / `tcrossprod()`

\[
\begin{align*}
crossprod(A) & := A^T A = A \cdot A^T = t(A) \%\% A \\
tcrossprod(A) & := A A^T = A \cdot A^T = A \%\% t(A) = \text{crossprod}(t(A))
\end{align*}
\]

where both return a symmetric matrix, by definition, and both use fast algorithms internally (LAPACK for dense, CHOLMOD for sparse):

\[
> \text{identical(tcrossprod(A), crossprod(t(A)))}
\]

[1] TRUE

\[
> \text{tcrossprod(A)}
\]

10 x 10 sparse Matrix of class "dsCMatrix"

\[
\begin{bmatrix}
1, 49 & . & . & . & . & . & . & . & . & . \\
2, . & 12600 & 7200 & 2700 & . & . & . & . & . & . \\
3, . & 7200 & 12796 & 7200 & 2700 & . & . & . & . & . \\
4, . & 2700 & 7200 & 13041 & 7200 & 2700 & . & . & . & . \\
5, . & . & 2700 & 7200 & 21484 & 12600 & 5400 & . & . & . \\
6, . & . & . & 2700 & 12600 & 25525 & 14400 & . & . & . \\
7, . & 2700 & 588 & . & 5400 & 14400 & 18864 & . & . & . \\
8, . & . & . & . & 2401 & . & . & . & . & . \\
9, . & . & . & . & . & . & . & . & . & . \\
\end{bmatrix}
\]

Now we have a symmetric matrix: `dsCMatrix`.

S4 Example: Matrix Classes

▶ How can we design a class or family of classes to hold matrices?
▶ Do we need two separate classes for positive-definite and general symmetric matrices; what about sparse or logical matrices, ...?
▶ What should happen if we get a general matrix but need a lower triangular one for some computations?

Classes and Objects

▶ A class is a static entity written as program code designed to represent objects of a certain type using slots (which in turn have other classes etc.)
▶ The class defines how an object is represented in the program.
▶ An object is an instance of the class that exists at run time.
Inheritance

▶ The hierarchical structure of classes we have seen for Matrices is very typical for object-oriented programming.
▶ Classes dMatrix and TsparseMatrix extend class Matrix by defining additional slots, they inherit from Matrix.
▶ A class inheriting from another class must have all slots from the parent class, and may define additional new slots.

Methods

▶ Once the classes are defined we probably want to perform some computations on objects. E.g., a natural operation for Matrices is to apply arithmetic, subset, use chol() etc.
▶ In most cases we don’t care whether the computer internally stores the Matrix in sparse or dense format, the computer should decide how to perform the task.
▶ The S way of reaching this goal is to use generic functions and method dispatch: the same function performs different computations depending on the types (i.e., classes) of its arguments (→ “signature”).

The S4 system

▶ define classes: setClass()
▶ create objects: new()
▶ define generics: setGeneric()
▶ define methods: setMethods()
▶ convert objects: as(), setAs()
▶ check object validity: setValidity(), validObject()
▶ access registry: getClass(), showMethods(), getMethod()
▶ ...

R Package Matrix: Compelling reasons for S4

1. Classes for Matrices: well-defined inheritance hierarchies:
   1.1 Content kind: Classes dMatrix, lMatrix, nMatrix, (iMatrix, zMatrix) for contents of double, logical, pattern (and not yet integer and complex) Matrices), where nMatrix only stores the location of non-zero matrix entries (where as logical Matrices can also have NA entries)
   1.2 sparsity: denseMatrix, sparseMatrix
   1.3 structure: general, triangular, symmetric, diagonal Matrices
2. Inheritance: Visualisation via graphs
3. **Multiple** Inheritance
**Multiple Dispatch in S4 .... for Matrix operations**

Methods for "Matrix"-matrices: Often 2 matrices involved..
1. \( x \%*\% y \)
2. `crossprod(x,y) ← x^T y`
3. `tcrossprod(x,y) ← xy^T`
4. \( x + y ← "Arith" \) group methods
5. \( x ≤ y ← "Compare" \) group methods

and many more: Dispatch happens according to Classes of both (or even more) arguments; S3 ("ops.Matrix") can only dispatch according to first argument (= “single dispatch”).

**Goals of Matrix package**

1. Interface to *LAPACK* = state-of-the-art numerical linear algebra for dense matrices
   - making use of special structure for symmetric or triangular matrices (e.g. when solving linear systems)
   - setting and keep such properties allows more optimized code in these cases.
2. Sparse matrices for large designs: regression, mixed models, etc
3. . . . . [omitted in this talk]

Hence, quite a few different classes for matrices.

**3-way Partitioning of “Matrix space”**

Logical organization of our Matrices: Three (3) main “class classification” for our Matrices, i.e., three “orthogonal” partitions of “Matrix space”, and every Matrix object’s class corresponds to an intersection of these three partitions.

i.e., in R’s S4 class system: We have three independent inheritance schemes for every Matrix, and each such Matrix class is simply defined to contain three virtual classes (one from each partitioning scheme), e.g.

```r
setClass("dgCMatrix", contain=c("CsparseMatrix", "dsparseMatrix", "generalMatrix"), validity= function(..) { .... a bit messy ... })
```

```r
> library(Matrix)
> length(allCl <- getClasses("package:Matrix"))
[1] 98

> ## Those called "...Matrix":
> length(M.CI <- grep("Matrix\$", allCl, value = TRUE))
[1] 70

i.e., *many* ..., each inheriting from root class "Matrix"

```r
> str(subs <- showExtends(getClassDef("Matrix")$subclasses, + printTo=FALSE))
List of 2
$ what: chr [1:76] "compMatrix" "triangularMatrix" "dMatrix" "iMatrix"...
$ how : chr [1:76] "directly" "directly" "directly" "directly" ...
```

```r
> ## even more... : All those above and these in addition:
> subs$what[ ! (subs$what %in% M.CI)]
[1] "Cholesky" "pCholesky" "BunchKaufman"
[4] "pBunchKaufman"
```

```r
many Matrix classes ... 
> library(Matrix)
> length(allCl <- getClasses("package:Matrix"))

[1] 98

> ## Those called "...Matrix":
> length(M.CI <- grep("Matrix\$", allCl, value = TRUE))
[1] 70

i.e., *many* ..., each inheriting from root class "Matrix"

```r
> str(subs <- showExtends(getClassDef("Matrix")$subclasses, + printTo=FALSE))
List of 2
$ what: chr [1:76] "compMatrix" "triangularMatrix" "dMatrix" "iMatrix"...
$ how : chr [1:76] "directly" "directly" "directly" "directly" ...
```

```r
> ## even more... : All those above and these in addition:
> subs$what[ ! (subs$what %in% M.CI)]
[1] "Cholesky" "pCholesky" "BunchKaufman"
[4] "pBunchKaufman"
```
... a bit messy ...
The three partitioning schemes are:

1. **Content type**: Classes `dMatrix`, `lMatrix`, `nMatrix` (and not yet `iMatrix`, `zMatrix`) for entries of type `double`, `logical`, `pattern` (and not yet `integer` and `complex`) Matrices. `nMatrix` only stores the location of non-zero matrix entries (whereas logical Matrices can also have `NA` entries!)

2. **Structure**: general, triangular, symmetric, diagonal Matrices

3. **Sparsity**: `denseMatrix`, `sparseMatrix`

First two schemes: a slight generalization from LAPACK for dense matrices.

**3D space of Matrix classes**

Three-way partitioning of Matrix classes visualized in 3D space, dropping the final `Matrix`, e.g., "d" instead of "dMatrix":

```r
> clGrid <- expand.grid(dim1 = d1, dim2 = d2, dim3 = d3, + KEEP.OUT.ATTRS = FALSE)
> clGr <- data.matrix(clGrid)
> library(scatterplot3d)
> used for visualization:
```

**3-fold classification — Matrix naming scheme**

1. **"Actual" classes**: Matrix objects are of those; the above "points in 3D space"
   - "corMatrix" "ddiMatrix" "dgCMatrix" ...

2. **Virtual classes**: e.g. the above coordinate axes categories. Superclasses of actual ones cannot have objects of, but importantly many methods for these virtual classes.

Actual classes follow a "simple" terse naming convention:

```r
> str(M3cl <- grep("\^\w*[a-z]*\w*\$",M.Cl, value = TRUE))
> substring(M3cl,1,3)
```

```r
corMatrix" "ddiMatrix" "dgCMatrix" ...
```

```r
> substr(M3cl,1,3)
```

```r
[1] "cor" "ddi" "dgC" "dge" "dgR" "dgT" "dpo" "dpp" "dsC" "dsp"
[11] "dsR" "dsT" "dsc" "dtC" "dtP" "dtR" "dTR" "ldi" "lgC"
[21] "lge" "lgR" "lgT" "lsC" "lsP" "lsR" "lst" "lsy" "ltC" "ltP"
[31] "ltp" "ltp" "ltp" "ltp" "ltp"
```

```r
> M3cl <- M3cl[M3cl != "corMatrix"] # corMatrix not desired in final use
```
Matrix 3d space: filled (2)

Matrix 3d space: filled (3)

Matrix 3d space: filled (4)

Defining S4 Classes

```r
## a small subset of ..../Matrix/R/AllClass.R

## Virtual Classes

# Virtual class of all Matrix objects
setClass("Matrix",
  representation(Dim = "integer", Dimnames = "list", "VIRTUAL"),
  prototype = prototype(Dim = integer(2), Dimnames =
    validity = function(object) {
      Dim <- object@Dim
      if (length(Dim) != 2)
        return("Dim slot must be of length 2")
      ...
    })
)

# Virtual class of numeric matrices
setClass("dMatrix",
  representation(x = "numeric"), contains = "Matrix")

# Virtual class of logical matrices
setClass("lMatrix", representation(x = "logical", "VIRTUAL"))
```
## Object Conversion

For `image()`ing a sparse Matrix, we coerce it to a triplet matrix and then use the `lattice` package's

```r
levelplot(abs(x@x) ~ (x@j +1) * (x@i +1), ....)
```

```r
> data(CAex)
> str(CAex)
```

**Formal class 'dgCMatrix' [package "Matrix"] with 6 slots**

```r
..@ i  : int [1:216] 0 24 48 1 25 49 2 26 50 3 ...
..@ j  : int [1:73] 0 3 6 9 12 15 18 21 24 27 ...
..@ Dim : int [1:2] 72 72
..@ Dimnames:List of 2
.. ..$ : NULL
.. ..$ : NULL
..@ x  : num [1:216] 0.999998 0.000132 -0.000527 0.999999 ...
..@ factors : list()
```
Coercions (as(*, class) instead of "as.class(.)") happen automatically along inheritance paths, and can (and should) be set up using setAs(.):

```r
## General rule: coercing to sparse : go to Csparse
setAs("Matrix", "sparseMatrix", function(from) as(from, "CsparseMatrix"))
setAs("CsparseMatrix", "symmetricMatrix", function(from) {
  if(isSymmetric(from)) { # then it's not triangular
    isTri <- is(from, "triangularMatrix")
    if (isTri &amp; from@diag == "U")
      from <-.Call(Csparse.diagU2N, from)
    .Call(Csparse_general_to_symmetric, from, uplo = if(isTri) from@uplo else "U")
  } else
    stop("not a symmetric matrix; consider forceSymmetric()")
}
```

Validity Checking

S4 objects are automatically checked for correct types whenever a slot is modified:

```r
> M@x <- "Hello"
```

Error in checkSlotAssignment(object, name, value) :
Assignment of an object of class "character" is not valid for slot "x" in an object of class "dgeMatrix";
......

In addition, a ("validity") function can be defined which checks whether an object is valid, e.g.,

```r
setValidity("Matrix", function(object) {
  Dim <- object@Dim
  if (length(Dim) != 2)
    return("Dim slot must be of length 2")
  if (any(Dim < 0))
    return("Dim slot must contain non-negative values")
  Dn <- object@Dimnames
  if (!is.list(Dn) || length(Dn) != 2)
    return("'Dimnames' slot must be list of length 2")
  # 'else' ok :
  TRUE
})
```
Of course one could simply use functions to check object validity, the advantages of `setValidity()` are:

- Validity checking methods are stored together with class definitions.
- If slots are themselves objects of classes with validity checks, they are also (recursively) checked.

```
Registry – Methods
> showMethods("show") # too many!
Function: show (package methods)
  object="ANY"
  object="BunchKaufman"
  object="classRepresentation"
  object="denseMatrix"
  object="diagonalMatrix"
  object="dsyMatrix"
  object="dtrMatrix"
  object="genericFunction"
  object="MatrixFactorization"
  object="MethodDefinition"
  object="MethodWithNext"
  object="ObjectsWithPackage"
  object="pBunchKaufman"
  object="signature"
  object="sparseMatrix"
  object="sparseVector"
  object="traceable"
```

```
Specific Method

> # Give method which will be dispatched for show(M) : 
> selectMethod("show", "dgeMatrix")
Method Definition:

  function (object)
  prMatrix(object)
  <environment: namespace:Matrix>

  Signatures:
  object
  target "dgeMatrix"
  defined "denseMatrix"

  More tools to explore methods:
  selectMethod(), existsMethod(), hasMethod(), ...
```

```
A special case is to turn S3 generics to S4:
> boxplot
function (x, ...)
UseMethod("boxplot")
<environment: namespace:graphics>

> setGeneric("boxplot")
[1] "boxplot"

> boxplot

standardGeneric for "boxplot" defined from package "graphics"

function (x, ...)
standardGeneric("boxplot")
<environment: 0x87400f8>
Methods may be defined for arguments: x
Use showMethods("boxplot") for currently available ones.
```
Multiple Dispatch

S3 methods have to do a lot of if()... else... computations to check what their arguments actually are:

```r
> graphics:::plot.factor
function (x, y, legend.text = levels(y), ...)
{
  if (missing(y) || is.factor(y)) {
    ...
  }
  if (missing(y)) {
    barplot(table(x), axisnames = axisnames, ...)
  } else if (is.factor(y)) {
    barplot(table(y, x), legend.text = legend.text, axisnames = axisnames,
    ...
  } else if (is.numeric(y))
    boxplot(y ~ x, ...)
  else NextMethod("plot")
}
```

Using dispatch on *multiple* arguments makes the code much more transparent:

```r
plot( x , y ) — 2 argument dispatch
setMethod("plot", signature(x = "numeric", y = "factor"),
  function(x, y, ...) boxplot(x ~ y, horizontal = TRUE, ...
setMethod("plot", signature(x = "factor", y = "numeric"),
  function(x, y, ...) boxplot(y ~ x, horizontal = FALSE,
```

> with(iris, plot(Species, Sepal.Length))

```
> with(iris, plot(Species, Sepal.Length))
```

```
> with(iris, plot(Species, Sepal.Length))
```

> showMethods("plot")

```
> showMethods("plot")
```

```
> showMethods("plot")
```

> with(iris, plot(Sepal.Length, Species))

```
> with(iris, plot(Sepal.Length, Species))
```

```
> with(iris, plot(Sepal.Length, Species))
```

```
> with(iris, plot(Sepal.Length, Species))
```
Sparse Least Squares

Koenker and Ng (2003) were the first to provide a sparse matrix package for R, including sparse least squares, via `slm.fit(x,y, ...)`. They provide the following nice example of a model matrix (probably from a quantile smoothing context):

```r
> library(Matrix)
> data(KNex)  # Koenker-Ng example
> dim(KNex$mm)
[1] 1850  712
> print(image(KNex$mm, aspect = "iso", colorkey = FALSE, + main = "Koenker-Ng model matrix"))
```

or rather transposed, for screen display:

```
> X.X <- crossprod(KNex$mm)
> c1 <- chol(X.X)
> image(X.X, main= "X'X", aspect="iso", colorkey = FALSE, + main = "Koenker-Ng model matrix")
```

Cholesky for Sparse L.S.

For *sparse* matrices, the Cholesky decomposition has been the most researched factorization and hence used for least squares regression modelling. Estimating $\beta$ in the model $y = X\beta + \epsilon$ by solving the *normal equations*

$$X^TX\beta = X^Ty$$  \hspace{1cm} (1)

the Cholesky decomposition of the (symmetric positive semi-definite) $X^TX$ is $LL^T$ or $R^TR$ with lower-left upper-right triangular matrix $L$ or $R \equiv L^T$, respectively.

System solved via two triangular (back- and forward-) “solves”:

$$\hat{\beta} = (X^TX)^{-1}X^Ty = (LL^T)^{-1}X^Ty = L^{-T}L^{-1}X^Ty$$  \hspace{1cm} (2)

CHOLMOD (Tim Davis, 2006): Efficient sparse algorithms.

Cholesky – “Fill-in”

The usual Cholesky decomposition works, ...

```r
> X.X <- crossprod(KNex$mm)
> c1 <- chol(X.X)
> image(X.X, main= "X'X", aspect="iso", colorkey = FALSE, + main = "Koenker-Ng model matrix")
```

but the resulting cholesky factor has suffered from so-called *fill-in*, i.e., its sparsity is quite reduced compared to $X^TX$. 

```
> image(c1, main= "chol(X'X)", ....)
```
Fill-reducing Permutation

So, chol(X'X) suffered from fill-in (sparsity decreased considerably).

Solution: Fill-reducing techniques which permute rows and columns of $X^\top X$, i.e., use $PX^\top XP'$ for a permutation matrix $P$ or in R syntax, $X.X[pvec,pvec]$ where $pvec$ is a permutation of 1:n.

The permutation $P$ is chosen such that the Cholesky factor of $PX^\top XP'$ is as sparse as possible

```
> image(t(c1), main= "t( chol(X'X) )", ........)
> c2 <- Cholesky(X.X, perm = TRUE)
> image(c2, main = "Cholesky(X'X, perm = TRUE)", ........)
```

(Note that such permutations are done for dense chol() when pivot=TRUE, but there the goal is dealing with rank-deficiency.)

Timing – Least Squares Solving

```
> y <- KNex$y
> m. <- as(KNex$mm, "matrix") # traditional (dense) Matrix
> system.time(cpod.sol <- solve(crossprod(m.), crossprod(m., y))
user system elapsed
2.885 0.028 2.927

> ## Using sparse matrices is so fast, we have to bump the time
> system.time(for(i in 1:10) ## sparse solution without permutation
+ sp1.sol <- solve(c1,solve(t(c1), crossprod(KNex$mm, y))
user system elapsed
0.056 0.000 0.055

> system.time(for(i in 1:10) ## sparse Cholesky with fill-reducing permutation:
+ sp2.sol <- solve(c2, crossprod(KNex$mm, y))
user system elapsed
0.012 0.000 0.143

> stopifnot(all.equal(sp1.sol, sp2.sol),
+ all.equal(as.vector(sp2.sol), c(cpod.sol)))
```

Conclusions

▶ S4 provides object oriented programming within an interactive environment.
▶ It can help you a lot to write clean and consistent code, and checks automatically if objects conform to class definitions.
▶ Multiple dispatch rather than nested if() ... else constructs in the body of functions.
▶ Fritz Leisch @ useR! 2004: I personally start all new packages using S4.