Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)
March 20–22, Vienna, Austria ISSN 1609-395X
Kurt Hornik, Friedrich Leisch & Achim Zeileis (eds.)
http://www.ci.tuwien.ac.at/Conferences/DSC-2003/

Converting a Large R Package to S4 Classes and Methods

Douglas M. Bates Saikat DebRoy

Abstract

The `nlme` package for fitting and examining linear and nonlinear mixed-effects models in R is a required package and also one of the largest R packages, based on source package size. In the first phase of a project to extend the capabilities of the nlme package to include generalized linear mixed models (glmm's), we reimplemented linear mixed-effects (lme) models using S4 classes and methods, as described in John Chambers' book "Programming with Data" and as implemented in the `methods` package for R. Our general goals for this phase are to incorporate new theoretical and computational developments for the lme model and to provide a faster, cleaner implementation of lme fits in R while including hooks for later extensions to the glmm model and the nlme model. In particular, we use our reStruct (random-effects structure) class in iterative PQL fits for glmm's, based on Brian Ripley's function `glmmPQL` from the `MASS` package.

As described in "Programming with Data", classes, slots and inheritance relationships must be declared explicitly when using the methods package. Although such formal declarations require package authors to be more disciplined than when using informal S3 classes, they provide assurance that each object in a class has the required slots and that the names and classes of data in the slots are consistent. This is important to us because we are trying to achieve both efficiency and flexibility. We provide flexibility by defining many classes and methods and by using multiple-argument signatures in method declarations. We achieve efficiency by implementing many methods in C code using the `.Call` interface and through liberal use of `GET_SLOT` and `SET_SLOT` within the C code.

We feel that the new implementation is much cleaner and easier to understand than the previous implementation, due in large part to the more extensive use of classes and methods. It is definitely faster and can handle larger problems than the previous implementation.

1 Introduction

The nlme package (Pinheiro and Bates, 2000) for fitting and examining linear and nonlinear mixed-effects models is a large R package. It consists of more than 500 R functions, 3500 lines of C code, and 40 data sets plus documentation and examples. The reason that there are so many functions in a package that is devoted to fitting just one general type of statistical model is to provide flexibility in specifying and examining the models. We also want to fit mixed-effects models efficiently.

To organize the large number of functions applied to different types of objects we created many classes of objects representing, for example, grouped data (groupedData), linear mixed-effects model structures (lmeStruct), nonlinear mixed-effects model structures, random-effects structures (reStruct), positive-definite parameterized matrices (pdMat), correlation structures (corStruct), variance functions (varFunc) and many different kinds of fitted models or summaries or plots derived from fitted models. Most of the 500 functions are methods for different classes of objects.

The computational methods described in Pinheiro and Bates (2000, ch. 2,7) for efficiently evaluating and profiling the log-likelihood or log-restricted-likelihood of a linear or nonlinear mixed-effects model with multiple, nested levels of random effects are quite formidable. Model matrices corresponding to the fixed-effects or to the random-effects terms in the statistical model are combined and decomposed repeatedly during the iterative optimization of the objective function to determine the parameter estimates. To achieve a reasonable level of efficiency in fitting models we coded the compute-intensive parts of the calculations in C.

We have begun a project to extend the capabilities of nlme to fit generalized linear mixed models (Raudenbush and Bryk, 2002, ch. 10), beginning with the method implemented by Brian Ripley in the glmmPQL function from the MASS package but also implementing estimation methods based on Laplacian and adaptive Gauss-Hermite approximations to the integral of the conditional density of the random effects.

In the first phase of this project we have reimplemented the data structures and computational algorithms for linear mixed models as S4 classes and methods. Our objectives for this reimplementation are:

- To encapsulate the underlying structures for linear mixed models in such a way that they can be extended to generalized linear mixed models and to nonlinear mixed models.
- To incorporate new theoretical and computational developments for the lme model. We have derived the analytic gradient of the profiled log-likelihood (or log-restricted likelihood) of a linear mixed model (DebRoy and Bates, 2003a,b) and have related the gradient results to an ECME (expectation conditional maximization either) optimization step. The analytic gradient allows for faster and, more importantly, more stable optimization.
- To convert the numerical linear algebra calls from Linpack and BLAS-1 calls to Lapack (Anderson, Bai, Bischof, Demmel, Dongarra, Croz, Greenbaum, Hammaring, McKenney, Ostrouchov, and Sorensen, 1992) and BLAS levels 1, 2, and 3. Lapack provides state-of-the-art algorithms and can provide a substantial performance boost when the ATLAS (Automatically Tuned Linear Algebra Software) implementations of the BLAS and some Lapack routines are available.

- To switch all calls of C code to the `.Call` interface so that entire R objects can be passed to and from the C code. This also allows direct access to the slots of S4 classed objects from within C code.
- To monitor the number of copies of objects that are created, especially those created within iterative algorithms. In §5 we discuss an example of a model fit to 375,000 observations on 135,000 subjects grouped into 3722 groups. The model matrices for the fixed effects can have as many as 40 or 50 columns, or about 150 MB for each copy of the model matrix and information derived from it. We need at least three arrays of this size to keep track of all the information we use. We do not want to create more than that if we can avoid doing so.

1.1 S3 versus S4 classes and methods

Object-oriented programming is a powerful tool for organizing the representation of information (classes) and the actions that are applied to these representations (methods). A system of classes and methods for the S language was introduced in Chambers and Hastie (1992). We will call this the S3 class system, to distinguish it from the S4 class system described in Chambers (1998) and implemented for R in the `methods` package. Unlike object-oriented languages such as Java and C++ where methods are associated with a class definition, both the S3 and the S4 systems associate methods with the combination of a generic function and the classes of one or more of the arguments to that function.

S3 classes are informal: the class of an object is determined by its class attribute, which should consist of one or more character strings, and methods are found by combining the name of the generic function with the class of the first argument to the function. If a function having this combined name is on the search path, it is assumed to be the appropriate method. Classes and their contents are not formally defined in the S3 system - at best there is a "gentleman's agreement" that objects in a class will have certain structure with certain component names.

By contrast, S4 classes must be defined explicitly. The number of slots in objects of the class, and the names and classes of the slots, are established at the time of class definition. During computation with objects from the class they are validated against the definition. As in many other object-oriented systems, an S4 class can be declared to inherit from another class so S4 classes can be arranged in a hierarchy.

S4 also requires formal declarations of methods, unlike the informal system of using function names to identify a method in S3. An S4 method is declared by a call to `setMethod` giving the name of the generic and the "signature" of the arguments. The signature identifies the classes of one or more named arguments to the generic function. Special meta-classes named `ANY` and `missing` can be used in the signature.

S4 generic functions can be declared by a call to `setGeneric` or they can be automatically created by declaring a method for an existing function, in which case the function becomes generic and the current definition becomes the default method.

2 Package conversion: Creating S4 classes

The principle generic functions for mixed-effects models and the methods associated with them were already defined in version 3.1 of the nlme package. Although these generics and methods would be modified to some extent during the conversion to

S4 classes and methods, we could initially assume these would stay as they are and concentrate instead on determining what classes should be defined and how we should define them.

We could use the informal set of classes from the S3 version as a guide when formulating the S4 classes. We found, however, that we frequently reconsidered the structure of the classes during the conversion and usually ended up adding more slots to the classes than had been present in the informal, S3 version.

Consider, for example, the `pdMat` class of parameterized, positive definite, symmetric matrices. It is a virtual class, i.e. a class for which no objects can be created, but which is used to create a family of extended or derived classes. The matrices represented by objects that inherit from this class are determined by a non-redundant, unconstrained vector of parameters. In some parameterizations the dimensions of the matrix can be determined from the length of the parameter vector but in others, such as `pdIdent`, representing multiples of the identity, or `pdCompSymm`, representing matrices with compound symmetry, the parameter vector has a fixed length and the number of columns in the matrix must be stored separately. We decided to add an `Ncol` slot to all the `pdMat` classes and did so by declaring it in the virtual `pdMat` class. In fact, the `pdMat` class declares six slots that are automatically present in all classes inheriting from the `pdMat` class

```
setClass("pdMat",      # parameterized positive-definite matrices
  representation(form="formula",    # a model-matrix formula
                 Names="character", # column (and row) names
                 param="numeric",   # parameter vector
                 Ncol="integer",    # number of columns
                 factor="matrix",   # factor of the pos-def matrix
                 logDet="numeric"   # logarithm of the absolute value
                 ## of the determinant of the factor (i.e. half
                 ## the logarithm of the determinant of the matrix)
                 ),
  prototype(form=formula(NULL),
            Names=character(0),
            param=numeric(0),
            Ncol=as.integer(0),
            factor=matrix(numeric(0),0,0),
            logDet=numeric(0))
  )
```

After this definition most of the class definitions for other pdMat classes are trivial.

```
setClass("pdSymm", "pdMat")     # general symmetric pd matrices
setClass("pdScalar", "pdSymm")  # special case of positive scalars
setClass("pdLogChol", "pdSymm") # default parameterization
setClass("pdNatural", "pdSymm") # log sd and Fisher's z of correlation
setClass("pdMatrixLog", "pdSymm")# matrix logarithm parameterization
setClass("pdDiag", "pdMat")     # diagonal pd matrices
setClass("pdIdent", "pdMat")    # positive multiple of the identity
setClass("pdCompSymm", "pdMat") # compound symmetric pd matrices
```

It may seem odd or verbose to define all these classes that are trivial extensions of the `pdMat` and `pdSymm` virtual classes. The point of doing this is that, although the representations of these different classes have the same form, there are many operations that are specific to the classes and we can use specific methods for these operations.

Increasing the number of slots may be an inevitable consequence of revising the package (we tend to add capabilities more frequently than we remove them) but it may also be related to the fact that S4 classes must be declared explicitly and hence we consider the components or slots of the classes and the relationships between the classes more carefully.

3 Calling C functions with `.Call`

The `.Call` interface, through which a programmer can pass raw R objects to C code and receive raw R objects from the C code, has been part of R for several years. It was inspired by the `.Call` interface for S described in Chambers (1998). Several C macros for working with S4 classed objects, including `GET_SLOT`, `SET_SLOT`, `MAKE_CLASS` and `NEW`, all described in Chambers (1998) are are now available in R, or will be in R 1.7.0. (Note that the macro `NEW_OBJECT` is preferred to `NEW` when writing code for R only. These two macros have the same effect but `NEW_OBJECT` is less likely to conflict with other definitions.)

The combination of the formal classes of S4, the `.Call` interface, and these macros allows a programmer to manipulate S4 classed objects in C code nearly as easily as in R code. A common idiom is to have an S4 method call C code through the `.Call` interface. In the C code the values of slots are extracted with `GET_SLOT` and either modified in place or used to create slots for new objects. Such new objects are created and populated by calls to `MAKE_CLASS`, `NEW_OBJECT`, and `SET_SLOT`.

Because the C code is called from a method, the programmer can be confident of the classes of the objects passed in the call and the classes of the slots of those objects. Much of the checking of classes or modes and possible coercion of modes that is common in C code called from R can be bypassed.

We found that we would initially write methods in R then translate them into C if warranted. The nature of our calculations, frequently involving multiple decompositions and manipulations of sections of arrays, was such that the calculations could be expressed in R but not very cleanly. Once we had the R version working satisfactorily we could translate into C the parts that were critical for performance or were awkward to write in R. An important advantage of this mode of development is that we could use the same slots in the C version as in the R version and create the same types of objects to be returned.

We feel that defining S4 classes and methods in R then translating parts of method definitions to C functions called through `.Call` is an extremely effective mode for numerical computation. Programmers who have experience working in C++ or Java may initially find it more convenient to define classes and methods in the compiled language and perhaps define a parallel series of classes in R. (We did exactly that when creating an early version of the Matrix package for R.) We encourage such programmers to try instead this method of defining only one set of classes, the S4 classes in R, and use these classes in both the interpreted language and the compiled language.

3.1 Using replacement methods

The .Call interface is a powerful tool and, like many powerful tools, must be used carefully if you are to avoid hurting yourself with it. The programmer must be aware that the arguments are not copied when they are passed through .Call even though the semantics of R function calls require that arguments must not be modified by a function call. If you are to modify the value of an R object passed as an argument you must somehow copy its storage, usually by duplicating it or coercing it to another mode, before making any changes. Failure to do so can result in bugs that are extremely difficult to diagnose.

Duplicating or coercing R objects will usually require that the result be protected from the garbage collector by a call to the PROTECT macro. The effect of all calls to PROTECT must be undone by calling UNPROTECT before returning from the C function. Keeping track of what has been protected can sometimes be tedious.

There is one exception to the "don't modify the arguments" rule: a replacement function or a replacement method is allowed to modify its first argument. Because the class of the result is the same as the class of the first argument it is common to use this argument as the return value, after suitable modification of its contents. We found that we used replacement methods more frequently in our code than we had first expected. We tended to think in steps of creating an object then modifying it according to the values of other objects.

For example, a linear mixed-effects model is represented by an reStruct object. The core part of the code to fit such a model is

```
re <- reStruct(fixed = fixed, random = random,
               data = eval(mCall, parent.frame()),
               REML = method != "ML")
EMsteps(re) <- controlvals
LMEoptimize(re) <- controlvals
```

where we construct the reStruct object, perform some number of EM updates on it then perform general nonlinear optimization on it.

4 Use of Lapack and ATLAS

Linpack and Eispack routines have been used for numerical linear algebra in R since its inception and are part of the R API. The Linpack and Eispack packages have been largely superseded by Lapack (Anderson et al., 1992) which provides, in some cases, better algorithms and, in nearly all cases, more effective implementations of the algorithms. Some of the effectiveness of the implementations, especially for large arrays, comes from more extensive use of the Basic Linear Algebra Subroutines (BLAS). As the name implies, these are basic routines for doing operations like multiplying two matrices or replacing $\boldsymbol{y}$ by $a\boldsymbol{x}+\boldsymbol{y}$. Even in sophisticated linear algebra algorithms, the majority of the numerical computation takes place in these basic operations, hence it is worthwhile devoting considerable effort to optimizing these routines. ATLAS (Automatically Tuned Linear Algebra Software) is a collection of highly optimized BLAS routines that can be compiled and optimized for different architectures. The combination of Lapack and ATLAS can give a considerable performance boost to algorithms that use numerical linear algebra extensively.

The R Core Development Group (primarily Brian Ripley) has been migrating R from Linpack and Eispack to Lapack. Beginning with R-1.7.0 the double precision

Lapack routines and some of the double precision complex Lapack routines will be part of the R API. We converted all the linear algebra in the lme calculations to Lapack, with gratifying results as described in the next section.

5 Timing results

Rodriguez and Goldman (1995) simulated 100 sets of 2445 binary responses grouped into 1558 families in 161 communities and fit generalized linear mixed models with two levels of random effects to these. The implementation of `glmmPQL` in the new version of the package is roughly 5 times as fast on these fits as the previous version that used repeated calls to `lme`.

We also fit a linear mixed-effects model to 378047 mathematics scores of 134713 students on the Texas Assessment of Academic Skills (TAAS). The data were all the test scores of students in grades 3 to 8 in Dallas, Texas during 1994 to 2000. The particular model that we fit had a fixed-effects vector of length 47, resulting in very large model matrices. A fit with the previous version of the lme function took 993 seconds of user time (1093 seconds elapsed time) on a 2.0 GHz Pentium-4 machine with 1.0 GB of PC-2700 memory running Debian GNU/Linux. The new version of lme took 221 seconds user time (345 seconds elapsed time) on the same machine.

6 Conclusions

We feel that we have met our objectives in reimplementing the lme part of the `nlme` package using S4 classes and methods. Although the performance boost from using Lapack and ATLAS is gratifying, we feel that the biggest gain is in making the code much cleaner and easier to understand and in exposing interfaces that can be used by models that extend the linear mixed-effects model.

Code clarity is enhanced by the fact that S4 classes and the `.Call` interface allow programmers to work with the same class definitions in R code and in C code. We have also found that liberal use of replacement functions and methods allows us to maintain control of the number of copies of objects being created.

Acknowledgements

This work is supported by U.S. Army Medical Research and Materiel Command under Contract No. DAMD17-02-C-0119. The views, opinions and/or findings contained in this report are those of the authors and should not be construed as an official Department of the Army position, policy or decision unless so designated by other documentation.

References

E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammaring, A. McKenney, S. Ostrouchov, and D. Sorensen. *Lapack Users' Guide.* SIAM, Philadelphia, 1992.

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

John M. Chambers and Trevor J. Hastie. *Statistical Models in S*. Chapman & Hall, London, 1992. ISBN 0-412-83040-X.

Saikat DebRoy and Douglas M. Bates. Computational methods for multiple level linear mixed-effects models. Technical Report 1076, Department of Statistics, University of Wisconsin-Madison, 2003a.

Saikat DebRoy and Douglas M. Bates. Computational methods for single level linear mixed-effects models. Technical Report 1073, Department of Statistics, University of Wisconsin-Madison, 2003b.

José C. Pinheiro and Douglas M. Bates. *Mixed-Effects Models in S and S-PLUS*. Springer, 2000. ISBN 0-387-98957-9.

Stephen W. Raudenbush and Anthony S. Bryk. *Hierarchical Linear Models: Applications and Data Analysis Methods*. Sage, second edition, 2002. ISBN 0-7619-1904-X.

Germán Rodriguez and Noreen Goldman. An assessment of estimation procedures for multilevel models with binary responses. *Journal of the Royal Statistical Society, Series A, General*, 158:73–89, 1995.

Affiliation

Douglas M. Bates, Saikat DebRoy
Department of Statistics
University of Wisconsin – Madison
E-mail: bates@stat.wisc.edu, saikat@stat.wisc.edu