The *coxme* function in S

Terry Therneau
Mayo Clinic

April 18, 2018

Contents

1 Introduction

The `coxme` function for mixed effects analyses based on a Cox proportional hazards model is one of the later additions to my survival functions in S [1]. It is easily the most complex bit of code in this ecosystem from all points of view: mathematical, algorithmic, and R code wise. As such, it seems like a natural candidate for documentation and maintainance using the literal programming model.

> Let us change or traditional attitude to the construction of programs. Instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *humans* what we want the computer to do. (Donald E. Knuth, 1984).

This document is my first foray into said domain. Time will tell if it is successful in creating both more reliable and better understood code.

Note that almost all of the code uses a *.Rnw* suffix, taking advantage of the very capable emacs modes that are part of the ESS pacakge. However, it is not processed by the `Sweave` or `knitR` packages, both of which are designed for reports containing *exectuted* S code. In contrast this document makes use of the `noweb` package to document (weave) and generate the R source code (tangle) from a single file. The `noweb` weave command is considerably simpler than that for `knitr` since it does not have to evaluate code, while its tangle function is more complex.

2 Main program

The `coxme` code starts with a fairly standard argument list.

```
⟨coxme⟩=
 coxme <- function(formula,  data,
         weights, subset, na.action, init,
         control, ties= c("efron", "breslow"),
         varlist, vfixed, vinit,
         x=FALSE, y=TRUE,
         refine.n=0, random, fixed, variance,  ...) {

     #time0 <- proc.time()    #debugging line
     ties <- match.arg(ties)
     Call <- match.call()

     ⟨process-standard-arguments⟩
     ⟨decompose-formula⟩
     ⟨build-control-structures⟩
     ⟨call-computation-routine⟩
     ⟨finish-up⟩
 }
```

[1] S is a statistical language with R and S-Plus being implementations of that language. I use R daily, but still refer to my code as S

The arguments to the function are described below, omitting those that are identical to the `coxph` function.

formula The formula desribing the fixed and random effects. This will be discussed in detail below.

varlist An optional list, with one element per random term, that describes the variance structure of the random effects. It need not be a list if there is only one random term.

vfixed An optional list (or vector) of fixed values for selected variance components.

vinit Initial value(s) for the variance components in the iteration.

refine.n The number of Monte Carlo iterations to be done at the final iteration, to refine the Laplace approximation of the likelihood.

random, fixed, variance These are included for backwards compatability with the first verion of coxme. They may be removed at some point in the future.

The `sparse` option has been moved to the coxme.control function. Cox model variance matrices are never sparse, but we have found that in one very particular instance we can ignore many off-diagonal elements. This combined with the naturally sparse structure of the penalty matrix can lead to substantial reductions in computational time. The ignorable elements arise for a random intercept term. In this case the diagonal elements of the usual Cox model variance matrix are $O(p_i)$ and the off diagonals are $O(p_i p_j)$, where p_i is the fraction of subjects in group i. If both p_i and p_j are sufficiently small the corresponding off-diagonal may be effectively ignored. For a particular family study that motivated the code there were over twenty thousand subjects, with a random intercept per subject, and the computation was not feasable without this addition. Based on fairly limited experience, the lower level for the approximation is set at $p = 1/50$. The default values for the `sparse` option state that the approximation should only be used if there are > 50 levels for the grouping factor, and only for those groups representing .02 or less of the total. If there are multiple random effects only one is allowed a sparse representation, nor are random slopes ever represented in this way. Further reseach may reveal wider circumstances in which the approximation is workable, but for now only the one known case is allowed.

2.1 Basic setup

The `coxme` code starts its model handling with a nod to backwards compatability. The argument list starts with the usual `(formula, data, weights, ...` set, but also allows `random` and `fixed` as optional arguments. If they are present, it assumes that someone is using the old style, and glues the fixed and random parts together into a single formula. Because the old form had `fixed` as its first argument, we also need to allow for the case where the user has assumed that the first, unnamed argument to the call, which now maps to the `formula` argument, is the fixed portion.

The other change was that the old `variance` argument became `vfixed`. This set of names makes a lot mores sense for the user as it is now in parallel with `vinit`.

⟨*process-standard-arguments*⟩=
```
 if (!missing(fixed)) {
     if (missing(formula)) {
```

```
        formula <- fixed
        warning("The 'fixed' argument of coxme is depreciated")
        }
    else stop("Both a fixed and a formula argument are present")
    }
if (!missing(random)) {
    warning("The random argument of coxme is depreciated")
    if (class(random) != 'formula' || length(random) !=2)
        stop("Invalid random formula")
    j <- length(formula)   #will be 2 or 3, depending on if there is a y

    # Add parens to the random formula and paste it on
    formula[[j]] <- call('+', formula[[j]], call('(', random[[2]]))
    }

if (!missing(variance)) {
    warning("The variance argument of coxme is depreciated")
    vfixed <- variance
    }
```

A formula in S is represented as a list of length 2 or 3, whose first element is as.name(' '), then the left hand side, if present, then the right hand side. Note that the old version of coxme contains almost the same code, since to correctly handle missing values it needed to retrieve all the relevant variables, both fixed and random, with a single list.

The program then executes a fairly standard step to retrieve the model frame. The model.frame function does not correctly handle vertical bars in a random term, the subbar function replaces each of these with a '+'.

⟨*process-standard-arguments*⟩=

```
temp <- call('model.frame', formula= subbar(formula))
for (i in c('data', 'subset', 'weights', 'na.action'))
    if (!is.null(Call[[i]])) temp[[i]] <- Call[[i]]
if (is.R()) m <- eval.parent(temp)
else        m <- eval(temp, sys.parent())
```

The final line is one of the few in the code that is specific to the particular S engine being used.

One question that comes up when first seeing this code, is "why not the simpler code"

```
temp <- model.frame(formula, data=data, subset=subset,
                    weights=weights, na.action=na.action)
```

The answer is that if any of the optional arguments were missing, then we would get an error. What the code above does is to create the above call bit by bit. The starting point only includes the `formula` argument, which is required. Then any optional arguments that are actually present are copied over from `Call` (what the user typed) to the temp variable. Many older S functions take a different approach by the way. They first made a complete copy of the call, e.g. `temp <- Call`, and then remove arguments that they don't want `temp$ties <- NULL; temp$rescale <- NULL`

etc. I don't like this approach, since every time that a new argument is added to the function, we need to remember to also add it to this x-out list. Another alternate, found in much of the newer R code is

```
alist <- match(names(Call), c('formula','data', 'subset', 'weights',
                              'na.action')
temp <- Call[c(1, which(!is.na(alist)))]
temp\Verb!1! <- as.name('model.frame')
```

My code above automatically forces an error if the formula is missing.

The model frame that we have created will contain all the variables found in both the fixed and random portions of the model. The next step is a usual one — pull out special terms such as the response, offset, etc. Penalized terms are supported in `coxph` but are not allowed in `coxme` The most common penalized terms in `coxph` are frailty terms terms and psplines (smoothing splines, similar to generalized additive models). Frailty terms are simple shared random effects, it was an early way to get some of the functionality of `coxme` by grafting a new capability onto `coxph`. Pspline terms could be supported, in theory, in `coxme`, but the effort to do so appears daunting and it is left for some future coder.

⟨*process-standard-arguments*⟩=

```
Y <- model.extract(m, "response")
n <- nrow(Y)
if (n==0) stop("No observations remain in the data set")
if (!inherits(Y, "Surv")) stop("Response must be a survival object")
type <- attr(Y, "type")
if (type!='right' && type!='counting')
    stop(paste("Cox model doesn't support '", type,
                      "' survival data", sep=''))

weights <- model.weights(m)
if (length(weights) ==0) weights <- rep(1.0, n)
else if (any(weights <=0))
    stop("Negative or zero weights are not allowed")

offset <- model.offset(m)
if (length(offset)==0) offset <- rep(0., n)

# Check for penalized terms; the most likely is pspline
pterms <- sapply(m, inherits, 'coxph.penalty')
if (any(pterms)) {
    stop("You cannot have penalized terms in coxme")
    }

if (missing(control)) control <- coxme.control(...)
if (missing(init)) init <- NULL
```

2.2 Fixed effects

The mixed effects Cox model is written as

$$\begin{aligned} \lambda(t) &= \lambda_0(t) e^{X\beta + Zb} \\ b &\sim N(0, \Sigma(\theta)) \end{aligned}$$

The coefficient vectors β and b correspond the the fixed and random effects, respectively, with X and Z as the respective design matrices.

It is now time to build X, the design matrix for the fixed effects. We first separate the model into random and fixed effects terms using the `formula1` function. As an argument it takes the model formula as given by the user and it returns a list containing the fixed and random parts of the formula, respectively. If any vertical bars remain in the fixed result, then there is a problem with the supplied formula, usually a random effects term that was missing the enclosing parentheses.

⟨*decompose-formula*⟩=

```
flist <- formula1(formula)
if (hasAbar(flist$fixed))
    stop("Invalid formula: a '|' outside of a valid random effects term")

special <- c("strata", "cluster")
Terms <- terms(flist$fixed, special)
attr(Terms,"intercept")<- 1  #Cox model always has \Lambda_0
strats <- attr(Terms, "specials")$strata
cluster<- attr(Terms, "specials")$cluster
if (length(cluster)) {
    stop ("A cluster() statement is invalid in coxme")
    }
if (length(strats)) {
    temp <- untangle.specials(Terms, 'strata', 1)
    if (length(temp$vars)==1) strata.keep <- m[[temp$vars]]
    else strata.keep <- strata(m[,temp$vars], shortlabel=T)
    strats <- as.numeric(strata.keep)
    X <- model.matrix(Terms[-temp$terms], m)[,-1,drop=F]
    }
else X <- model.matrix(Terms, m)[,-1,drop=F]
```

The key tools for building the matrix are the `terms` and `model.matrix` functions, which are common to all S modeling routines. The `terms` function takes a standard formula, and returns an object that is used for later processing. The `specials` argument asks the function to note any calls to *cluster* or *strata* in the formula, this makes it possible for us to pull out those terms for special processing.

The *cluster()* function is used in `coxph` to obtain a generalized estimating equation (GEE) type of variance estimate. Random effects and GEE are two different ways to approach correlated outcomes, but they cannot be mixed. Thus such a term is invalid in a `coxme` model.

In a Cox model the baseline hazard λ_0 plays the role of an intercept, but the X matrix does not explicitly contain an intercept. Nevertheless, contrasts terms, such as the dummy variable codings for factors, need to be formed as though there were an intercept term. We thus mark the model as containing an intercept column by setting the intercept attribute of `terms` (and completely ignore any "-1" that the user put into the model) before calling `model.matrix`. After then remove the unneeded intercept column from the returned matrix. The resulting X matrix might have only one column; the `drop=F` option causes it to remain a matix and not become a vector. If there are only random effects in the model, X could even have 0 columns.

If there are strata, they are removed from the model formula before forming the X matrix, since strata effect only the baseline hazard. The variable `strata.keep` retains the strata levels as specified by the user. The variable `strats` has values of 1,2, ... and is simpler for the underlying C code to deal with.

2.3 Random effects

Creating the random effects components is more complicated than the fixed effects.

- We need to create both the Z matrix and Σ.
- The actual form of Z depends on the type of random effect, but often looks like the design matrix for a one way anova. There are many possible correlation structures Σ.
- If there are multiple random terms, each creates a block of columns in Z and block of values in Σ.
- For efficiency, any class variables in Z are represented in compressed form, i.e., random intercepts. Such variables are stored in a matrix F which has a single column for each class variable, with integer values of 1,2, ... that state which coefficient each observation contributes to.. Z will contain the remaining columns.

The basic flow of the routine is to process the random terms one at a time. The `varlist` argument describes a variance family for each term; and we do two calls for each. The first call is to the *initialize* member of the family, giving it the G containing the grouping variables along with covariates C and whether or not the left hand side contained an intercept, and appropriate portions of the initial values (vinit) or fixed variance (vfixed) specification. It returns corresponding columns of F, Z and a mapping `zmap` for each column of Z, a vector `itheta` containing the initial values for the non-fixed variance parameters (possibly on a transformed scale), and a private parameter list which will be passed forward to the matching `generate` and `wrapup` routines. Any transformation is private to the variance family function.

The `formula2` function is desribed later; it is responsible for further separating the components of each random terms for us: whether the left hand side has an intercept, any other variables on the left, grouping variables, and optional interaction.

Our first action is to check out the `varlist` option. This is complicated by the fact that users can give a partial one, or none, allowing the default to be used for other elements. In general `varlist` is a named list with one list element per random term. Each element of the list can be:

- A matrix or list of matrices. This is useful for genetic data in particular.

- A function which generates a coxme variance family object (of class `coxmevar`), or the result of a call to such a function.

We are not backwards compatable with all old-stlye `coxme` calls, in particular the use of the unevaluated `bdsI` function in a list. This was mostly used to generate models that can now be directly stated.

The first task is to decide which element of the list goes to which term. If the list is a collection of coxmevar objects, then they are used one by one and any remaining random terms get the default action of coxmeFull. If there is only one random term then `varlist` is not required to be a list of one element, but we immediately make it so. A single list of variance matrices can be ambiguous, for instance if there were 2 random terms and one list with two matrices: is this a pair of matrices for the first term, or one matrix for each? We force the user to resolve the ambiguity.

⟨*build-control-structures*⟩=
```
nrandom <- length(flist$random)
if (nrandom ==0) stop("No random effects terms found")
vparm <- vector('list', nrandom)
is.variance <- rep(TRUE, nrandom)  #penalty fcn returns a variance or penalty?
ismat <- function (x) {
    inherits(x, "matrix") || inherits(x, "bdsmatrix") | inherits(x, "Matrix")
}
if (missing(varlist) || is.null(varlist)) {
    varlist <- vector('list', nrandom)
    for (i in 1:nrandom) varlist[[i]] <- coxmeFull() #default
    }
else {
    if (is.function(varlist)) varlist <- varlist()
    if (class(varlist)=='coxmevar') varlist <- list(varlist)
    else if (ismat(varlist))
        varlist <- list(coxmeMlist(list(varlist)))
    else {
        if (!is.list(varlist)) stop("Invalid varlist argument")
        if (all(sapply(varlist, ismat))) {
            # A list of matrices
            if (nrandom >1)
                stop(paste("An unlabeled list of matrices is",
                           "ambiguous when there are multiple random terms"))
            else varlist <- list(coxmeMlist(varlist))
            }
        else {  #the user gave me a list, not all matrices
            for (i in 1:length(varlist)) {
                if (is.function(varlist[[i]]))
                    varlist[[i]] <-varlist[[i]]()
                if (ismat(varlist[[i]]))
                    varlist[[i]] <- coxmeMlist(list(varlist[[i]]))
                if (class(varlist[[i]]) != 'coxmevar') {
```

```
                    if (is.list(varlist[[i]])) {
                        if (all(sapply(varlist[[i]], ismat)))
                            varlist[[i]] <- coxmeMlist(varlist[[i]])
                        else stop("Invalid varlist element")
                        }
                    else stop("Invalid varlist element")
                    }
                }
            }
        }
    while(length(varlist) < nrandom) varlist <- c(varlist, list(coxmeFull()))
    }


if (!is.null(names(varlist))) { # put it in the right order
    vname <- names(varlist)
    stop("Cannot (yet) have a names varlist")
    indx <- pmatch(vname, names(random), nomatch=0)
    if (any(indx==0 & vname!=''))
        stop(paste("Varlist element not matched:", vname[indx==0 & vname!=''']))
    if (any(indx>0)) {
        temp <- vector('list', nrandom)
        temp[indx] <- varlist[indx>0]
        temp[-indx]<- varlist[indx==0]
        varlist <- temp
        }
    }

#check validity (result is never used)
check <- sapply(varlist, function(x) {
       fname <- c("initialize", "generate", "wrapup")
       indx <- match(fname, names(x))
       if (any(is.na(x)))
           stop(paste("Member not found in variance function:",
                      fname(is.na(indx))))
       if (length(x) !=3 || any(!sapply(x, is.function)))
           stop("Varlist objects must consist of exaclty three functions")
   })
```

At this point we have a valid `varlist` object, which is a list with one element per random term, each element is an object of class 'coxmevar'. The current options for these elements are

coxmeFull All variance/covariance terms between random elements are present. For instance the term `(1+age | group)` specifies a random intercept and slope. The variance structure will have 3 parameters: the variance of the intercepts, the variance of the slopes, and their covariance.

coxmeMlist The variance is assumed to be of the form $\sigma_1^2 A_1 + \sigma_2^2 A_2 + \ldots$ for a set of fixed matrices $A_1, A_2, \ldots$. This is commonly used in genetic studies where A_1 would be the kinship matrix for a set of subjects/families and A_2 might be the identity-by-descent (IBD) matrix for a particular locus.

other a user-defined varlist function.

Now we proceed through the list one element at a time, and do the necessary setup. The `itheta` vector will contain starting values for all of the variance parameters that are *not* fixed, and `ntheta[i]` the number of values that each random term contributed. The final θ vector will be null if all the parameters are fixed. The `ncoef` matrix has a row for each term, containing the number of intercepts and slopes for that term. The definition of 4 helper functions is deferred until later.

```
⟨build-control-structures⟩=
 ⟨get-cmat⟩
 ⟨get-groups⟩
 ⟨make-vinit⟩
 ⟨newzmat⟩
 fmat <- zmat <- matrix(0, nrow=n, ncol=0)
 ntheta <- integer(nrandom)
 ncoef  <- matrix(0L, nrandom, 2, dimnames=list(NULL, c("intercept", "slope")))
 itheta <-  NULL   #initial values of parameters to iterate over

 for (i in 1:nrandom) {
     f2 <- formula2(flist$random[[i]])
     if (f2$intercept & f2$group==1)
         stop(paste("Error in random term ", i,
                    ": Random intercepts require a grouping variable", sep=''))
     vfun <- varlist[[i]]
     if (!is.null(f2$interaction)) stop("Interactions not yet written")

     cmat <- getcmat(f2$fixed, m)
     groups <- getgroups(f2$group, m)
     ifun <- vfun$initialize(vinit[[i]], vfixed[[i]], intercept=f2$intercept,
                         groups, cmat, control)
     if (!is.null(ifun$error))
         stop(paste("In random term ", i, ": ", ifun$error, sep=''))
     vparm[[i]] <- ifun$parms
     if (!is.null(ifun$is.variance)) is.variance[i] <- ifun$is.variance
     itheta <- c(itheta, ifun$theta)
     ntheta[i] <- length(ifun$theta)

     if (f2$intercept) {
         if (!is.matrix(ifun$imap) || nrow(ifun$imap) !=n)
             stop(paste("In random term ", i,
                        ": Invalid intercept matrix F", sep=''))
```

```
        temp <- sort(unique(c(ifun$imap)))
        if (any(temp != 1:length(temp)))
            stop(paste("In random term ", i,
                       ": intercept matrix has an invalid element", sep=''))

        if (ncol(fmat) >0) fmat <- cbind(fmat, ifun$imap + max(fmat))
        else fmat <- ifun$imap
        ncoef[i,1] <- 1+ max(ifun$imap) - min(ifun$imap)
        }

    if (length(cmat)>0) {
        if (is.null(ifun$xmap) || is.null(ifun$X) ||
            !is.matrix(ifun$xmap) || !is.matrix(ifun$X) ||
            nrow(ifun$xmap) !=n || nrow(ifun$X) != n ||
            ncol(ifun$xmap) != ncol(ifun$X))
            stop(paste("In random term ", i,
                       "invalid X/xmap pair"))
        if (f2$intercept) xmap <- ifun$xmap - max(ifun$imap)
        else xmap <- ifun$xmap
        if (any(sort(unique(c(xmap))) != 1:max(xmap)))
             stop(paste("In random term ", i,
                       ": xmap matrix has an invalid element", sep=''))

        temp <- newzmat(ifun$X, xmap)
        ncoef[i,2] <- ncol(temp)
        zmat <- cbind(zmat, temp)
        }
}
if (any(is.variance) & !all(is.variance))
         stop("All variance terms must have the same is.variance setting")
```

The matrix F holds the columns associated with intercept terms, so has columns added only if the new random terms has a 1 on the left side of the formula. It is also (at present) the only case in which sparse computation is known to be valid. Further discussion of this rather subtle topic is found in the section on variance functions. The underlying C programs can't deal with holes in a factor variable. That is, every column of fmat must be integers, with minimum 1 and no gaps.

Now to fill in a few blanks from the above discussion. First the vector (list) of initial values `vinit` or fixed variance values `vfixed` given by the user may not be complete. We want to expand them out to to lists, and have the same length as `varlist`. In the case of multiple terms, we allow the user to specify a subset of them, using the names of the grouping variables. If names are not used (or are not unique) things match in order. If there is a single random term we allow a numeric vector. In the case of a single term someone might use a list, allow that too. The `list(unlist(vinit))` construct below might look odd, what it does is transform `list(sigma=.1)` to a list with element name "" (the name of the term) and whose first element is a vector of length 1 with an element name of "sigma", which is what the routine wants in the

end. That is, it looks like the result of `list(c(sigma=.1))`.

⟨*make-vinit*⟩=

```
 if (missing(vinit) || is.null(vinit)) vinit <- vector('list', nrandom)
 else {
     if (nrandom==1) {
         if (is.numeric(vinit)) vinit <- list(vinit)
         else if (is.list(vinit)) vinit <- list(unlist(vinit))
     }
     if (!is.list(vinit)) stop("Invalid value for `vinit' parameter")
     if (length(vinit) > nrandom)
         stop (paste("Vinit must be a list of length", nrandom))
     if (!all(sapply(vinit, function(x) (is.null(x) || is.numeric(x)))))
         stop("Vinit must contain numeric values")

     if (length(vinit) < nrandom)
         vinit <- c(vinit, vector('list', nrandom - length(vinit)))

     tname <- names(vinit)
     if (!is.null(tname)) {
         stop("Named initial values not yet supported")
         #temp <- pmatch(tname, names(flist$random), nomatch=0)
         #temp <- c(temp, (1:nrandom)[-temp])
         #vinit <- vinit[temp]
         }
   }

 if (missing(vfixed) || is.null(vfixed)) vfixed <- vector('list', nrandom)
 else {
     if (nrandom==1) {
         if (is.numeric(vfixed)) vfixed <- list(vfixed)
         else if (is.list(vfixed)) vfixed <- list(unlist(vfixed))
     }
     if (!is.list(vfixed)) stop("Invalid value for `vfixed' parameter")
     if (length(vfixed) > nrandom)
         stop (paste("Vfixed must be a list of length", nrandom))
     if (!all(sapply(vfixed,  function(x) (is.null(x) || is.numeric(x)))))
         stop("Vfixed must contain numeric values")

     if (length(vfixed) < nrandom)
         vfixed <- c(vfixed, vector('list', nrandom - length(vfixed)))

     tname <- names(vfixed)
     if (!is.null(tname)) {
         temp <- pmatch(tname, names(flist$random), nomatch=0)
         temp <- c(temp, (1:nrandom)[-temp])
```

```
        vfixed <- vfixed[temp]
        }
    }
```

The actual computation of the model is done in `coxme.fit`. This was separated from the main routine to leave the code in managable chunks.

⟨*call-computation-routine*⟩=
```
 fit <- coxme.fit(X, Y, strats, offset, init, control, weights=weights,
                  ties=ties, row.names(m),
                  fmat, zmat, varlist, vparm,
                  itheta, ntheta, ncoef, refine.n,
                  is.variance = any(is.variance))
```

Then we finish up by packaging up the results for a user. The first few lines are the case where a fatal error occured, in which case the result contains only the failure line. (Is this needed?)

⟨*finish-up*⟩=
```
 if (is.character(fit)) {
     fit <- list(fail=fit)
     oldClass(fit) <- 'coxme'
     return(fit)
     }
```

Now add labels to the fixed and random coefficients. The `coefficients` portion of the returned object conttains the values for $\hat{\beta}$ (fixed). and for the variances $\hat{\theta}$. The `frail` component contains the values for $\hat{b}$, a historical label.

⟨*finish-up*⟩=
```
 fcoef <- fit$coefficients
 nvar <- length(fcoef)
 if (length(fcoef)>0 && any(is.na(fcoef))) {
     vars <- (1:length(fcoef))[is.na(fcoef)]
     msg <-paste("X matrix deemed to be singular; variable",
                     paste(vars, collapse=" "))
     warning(msg)
     }
 if (length(fcoef) >0) {
     names(fit$coefficients) <- dimnames(X)[[2]]
     }
```

Add up the part of the linear predictor due to random terms, and add this to the fixed portion to get an overall linear predictor.

⟨*finish-up*⟩=
```
 rlinear <- rep(0., nrow(Y))
 if (length(fmat)) {
     for (i in 1:ncol(fmat)) {
         rlinear <- rlinear + fit$frail[fmat[,i]]
```

```
        }
    }
if (length(zmat)) {
    indx <- if (length(fmat)>0) max(fmat) else 0
    for (i in 1:ncol(zmat))
        rlinear <- rlinear + fit$frail[indx+i]*zmat[,i]
    }

if (nvar==0) fit$linear.predictor <- rlinear
else fit$linear.predictor <- as.vector(rlinear + c(X %*% fit$coef))
```

Our last action for the random terms is to call the `wrapup` functions, which retransform (if needed) θ back to the user's scale, re-insert (if needed) any fixed parameters, label the vector, and label/arrange the random coefficients b.

Intercept terms always come first in the random coefficients. In a model with `(trt|group) + (1|group)` on the right, the ncoef object will be

```
     intercept slope
[1,]         0     5
[2,]         5     0
```

which means that the first 5 elements of b= `fit$frail` belong to term2, and the second five to term 1. (Also that my example data had 5 levels for the group variable). The creation of `bindex` below depends on the fact that R stores matrices in row major order so it will go through the intercepts first and the other random coefficients second.

⟨*finish-up*⟩=

```
newtheta <- random.coef <- list()
nrandom <- length(varlist)
sindex <- rep(1:nrandom, ntheta) # which thetas to which terms
bindex <- rep(row(ncoef), ncoef) # which b's to which terms
for (i in 1:nrandom) {
    temp <- varlist[[i]]$wrapup(fit$theta[sindex==i], fit$frail[bindex==i],
                                vparm[[i]])
    newtheta <- c(newtheta, temp$theta)
    if (!is.list(temp$b)) {
        temp$b <- list(temp$b)
        names(temp$b) <- paste("Random", i, sep='')
        }
    random.coef <- c(random.coef, temp$b)
    }
fit$frail <- random.coef

fit$vcoef <- newtheta
fit$theta <- NULL
```

Last fill in a set of miscellaneous members of the structure

⟨*finish-up*⟩=
```
fit$n <- c(sum(Y[,ncol(Y)]), nrow(Y))
fit$terms <- Terms
fit$assign <- attr(X, 'assign')
fit$formulaList <- flist

na.action <- attr(m, "na.action")
if (length(na.action)) fit$na.action <- na.action
if (x)  {
    fit$x <- X
    if (length(strats)) fit$strata <- strata.keep
    }
if (y)     fit$y <- Y
if (!is.null(weights) && any(weights!=1)) fit$weights <- weights

fit$formula <- as.vector(attr(Terms, "formula"))
fit$call <- Call
fit$ties <- ties
names(fit$loglik) <- c("NULL", "Integrated", "Penalized")
oldClass(fit) <- 'coxme'
fit
```

2.4 Creating the C and F matrices

To create the columns for F there are 3 steps. First we get the variables from the data frame, treating each of them as a factor. This is then submitted to the appropriate coxme variance family function, which creates the integer matrix of codes that are actually used.

We extract the list of variable names for the nesting portion, at the same time checking that it consists of nothing but variables and slash operators.

⟨*get-groups*⟩=
```
getGroupNames <- function(x) {
    if (is.call(x) && x[[1]]==as.name('/'))
        c(getGroupNames(x[[2]]), getGroupNames(x[[3]]))
    else deparse(x)
    }

getgroups <- function(x, mf) {
    if (is.numeric(x)) return(NULL)  # a shrinkage effect like (x1+x2 | 1)
    varname <- getGroupNames(x)
    indx <- match(varname, names(mf), nomatch=0)
    if (any(indx==0)) stop(paste("Invalid grouping factor", varname[indx==0]))
    else data.frame(lapply(mf[indx], as.factor))
    }
```

A common task for the variance functions is to expand `school/teacher` type terms into a set of unique levels, i.e., to find all the unique combinations of the two variables. Teacher 1 in school 1

is not the same person as teacher 1 in school 2. We can't use the usual processing functions such as `model.matrix` to create the nesting variables, since it also expands the factors into multiple columns of a matrix. (This is how `lmer` does it.) We will use the `strata` function from the standard survival library.

⟨*expand.nested*⟩=
```
 expand.nested <- function(x) {
     xname <- names(x)
     x[[1]] <- as.factor(x[[1]])[,drop=T]
     if (length(x) >1) {
         for (i in seq(2, length(x), by=1)) {
             x[[i]] <- strata(x[[i-1]], x[[i]], shortlabel=TRUE, sep='/')
             xname[i] <- paste(xname[i-1], xname[i], sep='/')
             }
        }
     names(x) <- xname
     x
     }
```

Creation of the C matrix is just a bit more work. One issue is that none of the standard S contrast options is correct. With a Gaussian random effect, either a random intercept or a random slope, the proper constraint is $b'\Sigma = 0$; this is familiar from older statistics textbooks for ANOVA as the "sum constraint". For a random effect this constraint is automatically enforced by the penalized optimization, so the proper coding of a factor with k levels is as k indicator variables. We do this by imposing our own contrasts.

Update. I've realized that factors are more of a problem than I thought. The issue is that the downstream routine has to know when to use a common variance for two columns of cmat, and when not to. This means that it has to look at the and (even harder) that the correlation between an intercept and a factor is not clear. The variance function will examine the `assign` attribute of the model matrix to know which terms go together.

⟨*get-cmat*⟩=
```
 getcmat <- function(x, mf) {
     if (is.null(x) || x==1) return(NULL)
     Terms <- terms(eval(call("~", x)))
     attr(Terms, 'intercept') <- 0  #ignore any "1+" that is present

     varnames <-  attr(Terms, 'term.labels')
     ftemp <- sapply(mf[varnames], is.factor)
     if (any(ftemp)) {
         clist <- lapply(mf[varnames[ftemp]],
                         function(x) diag(length(levels(x))))
         model.matrix(Terms, mf, contrasts.arg =clist)
         }
     else model.matrix(Terms, mf)
     }
```

The initial function returns both a set of covariates X and a coefficient map for each column of X, showing for each row of data which coefficient the term maps to. If the map has multiple columns and/or any one of the columns has a lot of levels then Z can get very large. Additionally, if Z has p columns then the Hessian matrix for the corresponding parameters is p by p. This is an area where the code could use more sparse matrix intelligence, i.e., so that the expanded Z need never be created.

⟨*newzmat*⟩=
```
 newzmat <- function(xmat, xmap) {
     n <- nrow(xmap)
     newz <- matrix(0., nrow=n, ncol=max(xmap))
     for (i in 1:ncol(xmap))
         newz[cbind(1:n, xmap[,i])] <- xmat[,i]
     newz
     }
```

3 The model formula

3.1 Introduction

The first version of `coxme` followed the `lme` convention of using separate formulas for the fixed and random portions of the model. This worked, but has a couple of limitations. First, it has always seemed clumsy and unintuitive. A second more important issue is that it does not allow for variables that participate in both the fixed and random effects. The new form is similar (but not identical) to the direction taken by the `lmer` project. Here is a moderately complex example modivated by a multi-institutional study where we are concerned about possible different patient populations (and hence effects) in each enrolling institution.

```
coxme(Surv(time, status) ~ age + (1+ age | institution) * strata(sex))
```

This model has a fixed overall effect for age, along with random intercept and slope for each of the enrolling institutions. The study has a separate baseline hazard for males and females, along with an interaction between strata and the random effect. The fitted model will include separate estimates of the variance/covariance matrix of the random effects for the two genders. This is a type of model that could not be specified in the prior mode where fixed and random effects were in separate statements.

3.2 Parsing the formula

The next step is to decompose the formula into its component parts, namely the fixed and the random effects. The standard formula manipulation tools in R are not up to this task; we do it ourselves using primarily two routines called, not surprisingly `formula1` and `formula2`. The first breaks the formula into fixed and random components, where the fixed component is a single formula and the random component may be a list of formulas if there is more than one random term.

Formulas in S are represented as a parse tree. For example, consider the formula `y ~ x1 + x2*(x3 + x4)`. It's tree is shown in figure 1. At each level the figure lists the class of the object along with its

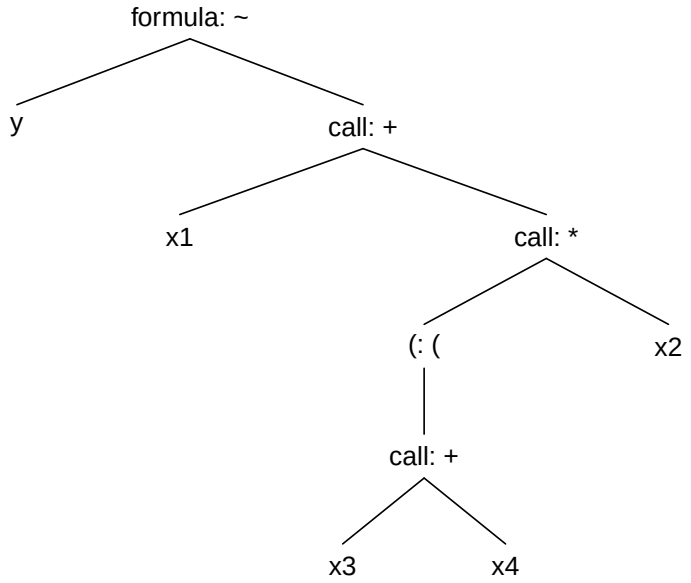

Figure 1: The parse tree for `y ~x1 + (x3 + x4)* x2`.

name; to lessen crowding in the plot objects of class 'name' do not have the class listed. The arguments to a call are the branches below each call. A formula is structured like a call to the "~" operator, and a parenthesised expression like a call with a single argument.

The `formula1` routine is called with the model formula, the response and the fixed parts are returned as the `fixed` component, the random parts are separated into a list. The primary concern of this function is to separate out the random terms; by definition this is a parenthesised term whose first child in the parse tree is a call to the vertical bar function. A random term is separated from the rest of the equation by one of the four operators +, -, *, or :, thus the parsing routine only has to worry about those four, anything else can safely be lumped into the fixed part of the equation.

We first deal with the top level call (the formula), and with parentheses. There are two cases. In the first, we have by definition found a random effects term. (The routine `formula2` will be used to check each random term for validity later). The second case is a random term found inside two sets of parentheses; this is redundant but legal. By simply passing on the list from the inner call the routine removes the extra set.

⟨*formula*⟩=
```
formula1 <- function(x) {
    if (class(x)=='formula') {  #top level call
        n <- length(x)  # 2 if there is no left hand side, 3 otherwise
        temp <- formula1(x[[n]])
        if (is.null(temp$fixed)) x[[n]] <- 1  # only a random term!
        else x[[n]] <- temp$fixed
        return(list(fixed=x, random=temp$random))
        }

    if (class(x) == '(' ) {
        if (class(x[[2]])== 'call' && x[[2]][[1]] == as.name('|')) {
            return(list(random = list(x)))
            }

        temp <- formula1(x[[2]])  # look inside the parenthesised object
        if (is.null(temp$fixed)) return(temp) #doubly parenthesised random
        else {
            # A random term was inside a set of parentheses, pluck it out
            #  An example would be (age + (1|group))
            if (length(temp$fixed) <= 2) x <- temp$fixed  #remove unneeded (
            else      x[[2]] <- temp$fixed
               return(list(fixed= x, random=temp$random))
            }
        }
```

Next we deal with the four operators one by one, starting with "+". We know that this call has exactly two arguments; the routine recurs on the left and then the right hand portions, and then merges the results. The merger has to deal with 5 cases, the left term either did or did not have a fixed effect term, and the right arm either does not exist, exists and does not have a random

effect, or exists without a random effect. The first case arises when someone accidentally has an extra sign such as `age + + sex + (1|grp)`; easy to do on a multi-line formula. We re-paste the two fixed effect portions together. The random terms are easier since they are lists, which concatonate properly even if one of them is null.

⟨*formula*⟩=
```
    if (class(x) == 'call' && x[[1]] == as.name('+')) {
        temp1 <- formula1(x[[2]])
        if (length(x)==2) return(temp1)  #no merge needed
        temp2 <- formula1(x[[3]])

        if (is.null(temp1$fixed)) {
            # The left-hand side of the '+' had no fixed terms
            return(list(fixed=temp2$fixed,
                        random=c(temp1$random, temp2$random)))
            }
        else if (is.null(temp2$fixed)) # right had no fixed terms
            return(list(fixed=temp1$fixed,
                        random=c(temp1$random, temp2$random)))
        else {
            return(list(fixed= call('+', temp1$fixed, temp2$fixed),
                        random=c(temp1$random, temp2$random)))
            }
        }
```

The code for "-" is identical except for one extra wrinkle: you cannot have a random term after a minus sign. Becase the expressions are parsed from left to right `~ age-1 + (1|group)` will be okay (though -1 makes no sense in a Cox model), but `~ age - (1 + (1|group))` will fail.

⟨*formula*⟩=
```
    if (class(x)== 'call' && x[[1]] == as.name('-')) {
        temp1 <- formula1(x[[2]])
        if (length(x)==2) return(temp1)
        temp2 <- formula1(x[[3]])
        if (!is.null(temp2$random))
            stop("You cannot have a random term after a - sign")

        if (is.null(temp1$fixed))  #no fixed terms to the left
            return(list(fixed=temp2$fixed,
                        random= temp1$random))
        else {  #there must be fixed terms to the right
            return(list(fixed= call('-', temp1$fixed, temp2$fixed),
                        random= temp1$random))
            }
       }
```

For the last line: we know there is something to the right of the '-', and it is not a naked random effects term, so it must be fixed.

Interactions are a bit harder. The model formula `~ (age + (1|group))*sex` for instance has an `age*sex` fixed term and a `(1|group)*sex` random term. Interactions between random effects are not defined. I don't know what they would mean if they were ....

⟨*formula*⟩=
```
    if (class(x)== 'call' && (x[[1]] == '*' || x[[1]] == ':')) {
        temp1 <- formula1(x[[2]])
        if (length(x) ==2) return(temp1)
        temp2 <- formula1(x[[3]])

        if (is.null(temp1$random) && is.null(temp2$random))
            return(list(fixed=x))   # The simple case, no random terms

        if (!is.null(temp1$random) && !is.null(temp2$random))
                stop ("The interaction of two random terms is not defined")
```

Create the new "fixed" term. In the case of `(1|group):sex`, there is no fixed term in the result. For `(1|group) *sex` the fixed term will be "sex". These are the two cases (and their mirror images) where only one of the left or right parts has a fixed portion. If both have a fixed portion then we glue them together.

⟨*formula*⟩=
```
        if (is.null(temp1$fixed) || is.null(temp2$fixed)) {
            if (x[[1]] == ':') fixed <- NULL
            else if (is.null(temp1$fixed)) fixed <- temp2$fixed
            else fixed <- temp1$fixed
            }
        else  fixed <- call(deparse(x[[1]]), temp1$fixed, temp2$fixed)
```

Create the new random term. The lapply is needed for `(((1|group) + (1|region)) * sex`, i.e., there are multiple groups in the random list. I can't imagine anyone using this, but if I leave it out they surely will and confuse the parser.

⟨*formula*⟩=
```
        if (is.null(temp2$random))  #left hand side was random
            random <- lapply(temp1$random,
                             function(x,y) call(':', x, y), y=temp2$fixed)
        else  #right side was
            random = lapply(temp2$random,
                                function(x,y) call(':', x, y), y=temp1$fixed)

        if (is.null(fixed)) return(list(random= random))
        else return(list(fixed=fixed, random=random))
        }
```

The last bit of the routine is for everything else, we treat it as a fixed effects term. A possible addition would be look for any vertical bars, which by definition are not a part of a random term — we've already checked for parentheses —and issue an error message. We do this instead in the parent routine.

⟨*formula*⟩=
```
    return(list(fixed=x))
 }
```

3.3 Random terms

Each random term is subjected to further analysis using the `formula2` routine. This has a lot of common code with `formula1`, since they both walk a similar tree. The second routine breaks a given random part into up to four parts, for example the result of `(1 + age + weight | region):sex` will be a list with elements:

- `intercept`: TRUE
- `variates`: age + weight
- `group`: region
- `interaction`: sex

We can count on `formula1` to have put any interaction term on the far right, which means that it will be the first thing we encounter.

⟨*formula*⟩=
```
 formula2 <- function(term) {
     if (is.call(term) && term[[1]] == as.name(':')) {
         interact <- term[[3]]
         term <- term[[2]]
         }
     else interact <- NULL

     if (class(term) != '(' || !is.call(term[[2]]) ||
                               term[[2]][[1]] != as.name('|'))
         stop("Formula error: Expected a random term")

     term <- term[[2]]  # move past the parenthesis
     out <- list(intercept=findIntercept(term[[2]]))
     out$group<- term[[3]]
     out$interaction <- interact
     out$fixed <- term[[2]]
     out
   }
```

This routine looks for an intercept term - that's all. It would be easiest to use the built in `terms` function for this, since the intercept could be anywhere, and someone might have put in a -1 term which makes it trickier. However, we can't: the default S strategy would claim that `(age+weight) |1)` has an intercept. As an advantage, we know that there can be no operators except "+" or "-" signs in the subformula at hand.

⟨*formula*⟩=
```
 findIntercept <- function(x) {
    if (is.call(x)) {
        if (x[[1]] == as.name('+')) findIntercept(x[[2]]) |findIntercept(x[[3]])
        else FALSE
        }
    else if (x==1) TRUE
         else FALSE
 }
```

3.4 Miscellaneous

Here is the simple function to look for any vertical bars. You might think of recurring on any function with two arguments, e.g., `if length(x)==3` on the fourth line. (The `findbars` routine in lmer, 3/2009, does this for instance, which shows that it must be a pretty sound idea, given the extensive use that code has seen.) However, that line would recur into other functions, like `logb(x5, 2)` for instance. The following is legal but has a vertical bar we wish to ignore: `I(x1 | x2)`. I have never seen an *actual* use of such a phrase, but nevertheless I'm taking the paranoid route.

⟨*formula*⟩=
```
 hasAbar <- function(x) {
   if (class(x)== 'call') {
         if (x[[1]]== as.name('|')) return(TRUE)
         else if (x[[1]]==as.name( '+') || x[[1]]== as.name('-') ||
                  x[[1]]==as.name( '*') || x[[1]]== as.name(':'))
             return(hasAbar(x[[2]]) || hasAbar(x[[3]]))
         else return(FALSE)
         }
     else if (class(x) == '(') return(hasAbar(x[[2]]))
     else return(FALSE)
     }
```

Here is a similar function which replaces each vertical bar with a '+' sign. This is needed for the `model.frame` call, which does not properly deal with vertical bars. Given a formula it returns a formula. We only recur on 4 standard operators to avoid looking inside functions. An example would be `~ age + I(x1 | x2) + (1|group)`; we take care not to look inside the `I()` or an ns() call, etc. I'm not sure that replacing the bar inside the I() function will cause any problems for model.frame; so I may be being overly cautious. The `if length(x)` statement below will most often arise from a formula with two + signs in a row. The second one is treated as unary so only has a single argument.

⟨*formula*⟩=
```
 subbar <- function(x) {
     if (class(x)=='formula') x[[length(x)]] <- subbar(x[[length(x)]])

     if (class(x)== 'call') {
```

```
    if (x[[1]]==as.name( '+') || x[[1]]== as.name('-') ||
        x[[1]]==as.name( '*') || x[[1]]== as.name(':')) {
        x[[2]] <- subbar(x[[2]])
        if (length(x)==3) x[[3]] <- subbar(x[[3]])
        }
    }
else if (class(x)== '(') {
    if (class(x[[2]])== 'call' && x[[2]][[1]] == as.name('|'))
        x[[2]][[1]] <- as.name('+')
    else x[[2]] <- subbar(x[[2]])
    }
x
}
```

4 Variance families

4.1 Structure

Each distinct random effects term corresponds to a distinct diagonal block in the overall penalty matrix, along with a set of penalized coefficients b. To make life easier for the maximizer, there may also be a transformation between the displayed variance coefficients and the internal ones, for instance a variance that is known to be > 0 will be maximized on the log scale. When there are multiple random terms in the formula then the `varfun`, `vinit`, and `variance` arguments must each be in the form of a list with one element per random term.

Variance family functions for `coxme` are similar in spirit to `glm` families: the functions set up the structure but do not do any work. Each of them returns a list of 3 functions, `initialize`, `generate`, and `wrapup`. Any optional arguments to the variance family are used to create these three; depending on the family they might apply to any one.

The initialize function is called with the X and G matrices for the given term, along with the sparse option and the appropriate vectors of initial and fixed values.

The return from a call to initialize is

itheta a list containing initial values for all the θ parameters that need to be optimized. Each element of the list will be a vector of values: the parent routine will try all combinations and then use the best as the starting value for the optim routine. If all the parameter values are fixed the list will be null.

imap the design matrix for random intercepts

X the covariate matrix for random slopes

xmap the design matrix for any random slopes

parm a list of arguments to be passed forward to the generate and wrapup functions.

error optional error message. This is passed up so the parent can print an error message with more information.

The input data `G` is a data frame with one variable per level of nesting. The G data passed in and the F matrix returned may not be the same. In particular, any class levels that are going to be treated as sparse will have been rearranged to so as to be the first columns of the penalty matrix (variance of the random effect), and so will have level indices of 1,2, ... In all the current routines the X matrix returned is identical to the X matrix input. In the future we may add scaling, however.

The *generate* function is called at each iteration with the current vector of estimates $\hat{\theta}$ and the appropriate parameter list. It will generate the variance matrix of the random effect, which may be either an ordinary matrix or a bdsmatrix. If there are multiple random effect terms, each of the generate functions creates its appropriate block.

The *wrapup* function is called when iteration is complete. Its job is to return the extended and re-transformed θ vector (fixed coefficients are re-inserted), and to format and label the vector of random coefficients.

For both initial values and fixed values we try to be as forgiving as possible, by first matching on names and then matching any unnamed arguments. Say for instance that the the term is `(1| race/sex)`, then all of

- `vinit = list(1,2)`
- `vinit = list(sex=2)`
- `vinit = list(sex=2, 1)`
- `vinit = c(sex=2, 1)`

are legal. We do this by augmenting pmatch to add in unnamed arguments. The `initmatch` function below will return a vector of integers of the same length as its input, showing which term they match to. So for the random term `(1| race/sex)` a user specification of `vinit=1:3` would return $(1, 2, 0)$ and `vinit=c(sex=2, school=3)` would give $(2, 0)$. The user cannot currently input a list of starting values: both `vfixed` and `vinit` allow only a single value per theta.

⟨*initmatch*⟩=
```
 initmatch <- function(namelist, init) {
     if (is.null(names(init))) iname <- rep('', length(init))
     else iname <- names(init)

     indx1 <- pmatch(iname, namelist, nomatch=0, duplicates.ok=TRUE)
     if (any(iname=='')) {
         temp <- 1:length(namelist)
         if (any(indx1>0)) temp <- temp[-indx1]   #already used
         indx2 <- which(iname=='')
         n <- min(length(indx2), length(temp))
         if (n>0) indx1[indx2[1:n]] <- temp[1:n]
         }
     indx1
     }
```

4.2 Sparseness

This is a good point to remind myself of an important distinction. When fitting the Cox likelihood we have to be aware of which terms of the partial likelihood's hessian matrix (second derivative) can be considered "sparse" or not. Because the C code expects the Hessian and the penalty to have exactly the same bdsmatrix form, the `kfun` function in `coxme.fit` has to orchestrate which parts of the penalty can be represented using the sparse part of a bdsmatrix (the blocks and blocksize components) and which has to use the dense part (the rmat component). Essentially, it treats terms 2, 3, ... as dense, and for the first term it *believes what the variance function sends it*. Thus, this is the point at which "sparseness" is determined.

A *b*lock *d*iagonal *s*ymmetric `bdsmatrix` object consists of two portions: a block diagonal section in the upper left and a dense border. Since it is symmetric only the right hand border is retained. If the block diagonal section has only a single block, then the matrix is dense; if there are many blocks it will be sparse.

Although the penalty matrices created by the variance function are themselves are often very sparse, the Cox model's Hessian matrix is never sparse. What we have found is that for some cases, one can pretend the Hessian is sparse, i.e., all of the terms in the block diagonal portion that are outside the blocks are considered zero.

4.3 coxmeFull

This is the default routine, which assumes a simple nested structure for the variance. Sparsity is assumed only for random intercepts, for those groups which have a small percent of the total.

The overall layout of the routine is below. It currently has only one optional parameter, which contols the form of nested effects

```
⟨coxmeFull⟩=
 coxmeFull <- function(collapse=FALSE) {
     collapse <- collapse
     # Because of R's lexical scoping, the values of the options
     #  above, at the time the line below is run, are known to the
     #  initialize function
     ⟨coxmeFull-init⟩
     ⟨coxmeFull-generate⟩
     ⟨coxmeFull-wrapup⟩
     out <- list(initialize=initialize, generate=generate, wrapup=wrapup)
     oldClass(out) <- 'coxmevar'
     out
     }
```

To describe the layout, we consider four cases of increasing complexity.

1. Shrinkage models, which have slopes but no groups `(x1+x2 | 1)`
2. A simple random intercept `(1|g1)`,
3. Nested random intercepts `(1 | g1/g2)`
4. Intercept and slopes, with or without nesting `(1 + x1 | g1/g2)`

There is also the invalid random effect `(1|1)`. Terms without either a covariate or an intercept to the left of the vertical bar have already failed with an error when the formula was parsed.

The `initialize` and `generate` routines each start by defining a few variables, and then treating the five cases one by one. The varinit and corinit values are the default lists of starting values to try for variance and correlation terms, respectively, and are defined in the coxme.control function. The current ones are rather ad hoc. We have found that the estimated standard deviation of a random effect is often between .1 and .3, corresponding to exp(.1)= 1.1 to exp(.3)= 1.35 fold "average" relative risks associated with group membership. This is biologically reasonable for a latent trait. A second common solution is a small random effect with 1–5% change in the hazard. (These will not be detectable, i.e., 'significant' unless the data set size is large of course.) Because we use the log(variance) as our iteration scale the 0–.001 portion of the variance scale is stretched out giving a log-likelihood surface that is almost flat; a Newton-Raphson iteration starting at log(.2) may have log(.0001) as its next guess and get stuck there, never finding a true maximum that lies in the range of .01 to .05. Finally, few data sets have solutions with variance > 1 for which a larger starting value suffices.

For the correlation we use 0 and .3. Negative values are not on the list since they can lead to an impossible (non positive definite) variance matrix. One cannot have 5 variables all with correlation -.3 for instance. There also appears to be less of a need for multiple starting estimates. Solutions rarely converge to the endpoints of the transformed range ($> .95$).

```
⟨coxmeFull-init⟩=
 initialize <- function(vinit, fixed, intercept, G, X,  control) {
     ngroup <- min(length(G), ncol(G))
     nvar   <- min(length(X), ncol(X))  # a NULL or a nx0 matrix yields 0
     sparse <- control$sparse
     ⟨initmatch⟩

     if (ngroup==0) {
         if (intercept)
             return(list(error=("Invalid null random term (1|1)")))
         else {
             ⟨coxmeFull-init-1⟩
             }
         }
     else {
         if (nvar==0) {
             ⟨initialize-inits⟩
             }

         # Deal with random slope terms
         if (ngroup ==1) {
             ⟨coxmeFull-init-2⟩
                 }
         else {
             if (collapse) {
                 ⟨coxmeFull-init-3b⟩
```

```
            }
        else {
            ⟨coxmeFull-init-3a⟩
            }
        }

    #Deal with slopes
    if (nvar > 0) {
        ⟨coxmeFull-init-4⟩
        }
    }
}
```

Case 1 of our initialize function will process a pure shrinkage term such as `(x1 + x2 | 1)`. In this case the two coefficients for `x1` and `x2` are considered to come from a Gaussian with a common variance σ^2. If the variance is fixed, this is equivalent to ordinary ridge regression.

First deal with initial values. There should be either 0 or 1 of them, named (if at all) with the first covariate. For the default starting values see the earlier discussion. These are then scaled by .5/std(X), the idea is that the defaults have proven to work well for the binomial indicator variables of a class variable, which usually have a std between 1/2 and 1/4. The variance matrix will be a diagonal, non-sparse, so after checking initial values there is almost nothing left to do.

⟨*coxmeFull-init-1*⟩=

```
xname <- dimnames(X)[[2]]
if (length(vinit) >0) {
  temp <- initmatch(xname[1], vinit)
  if (any(temp==0))
      return(list(error=paste('Element', which(temp==0),
                              'of initial values not matched')))
  else {
      if (vinit <=0) return(list(error="Invalid variance value, must be >0"))
      theta <- vinit
      }
  }
else theta <- control$varinit *.5 / mean(sqrt(apply(X,2,var)))

if (length(fixed) >0) {
    temp <- initmatch(xname[1], fixed)
    if (any(temp==0))
        return(list(error=paste('Element', which(temp==0),
                                'of fixed variance values not matched')))
    else theta <- fixed
    which.fixed <- TRUE
    if (theta <=0) return(list(error="Invalid variance value, must be >0"))
    }
else which.fixed <- FALSE
```

```
xmap <- matrix(0L, nrow=nrow(X), ncol=ncol(X))
for (i in 1:ncol(X)) xmap[,i] <- i

list(theta=list(log(theta))[!which.fixed], imap=NULL, X=X, xmap=xmap,
         parms=list(fixed=which.fixed, theta=theta[1],
                    xname=xname, case=1))
```

The generate function has a separate block for each of the 4 cases. To start, however, make sure that the exponential function never leads to a variance that is exactly zero or to a correlation of 1. The value 36 is close to `-log(.Machine$double.eps)`, used in the Splus care.exp function. For a coxph model, a variance ¿10 is usually pretty wild, and one less than .0001 is near 0 in behavior (for properly scaled variables), so this trucation does not affect any statistical properties of the estimates.

⟨*coxmeFull-generate*⟩=
```
generate= function(newtheta, parms) {
    theta <- parms$theta
    if (length(newtheta)>0) theta[!parms$fixed] <-
        exp(pmax(-36, pmin(36, newtheta)))

    if (parms$case==1) return(diag(length(parms$xname)) * theta)
    ⟨coxmeFull-generate-2⟩
    ⟨coxmeFull-generate-3⟩
    ⟨coxmeFull-generate-4⟩
    }
```

Case 2 is the simple one of a single grouping variable and no covariates. If sparseness applies, then the levels of the variable are reordered to put the infrequent levels first, and the variance matrix starts with nsparse 1×1 blocks. The input will have G as a single column data frame containing a single grouping variable, often represented as a factor and X will be null. If G has g levels, then the vector of random intercepts will be of length g, there is a single random variance, and

$$b_i \sim N(0, \sigma^2 I)$$

Several times in the code we make use of the fact that matrices are stored in column major order. Thus a sequence of indices `i, i+ ncol(R)+1, i+ 2*(ncol(R)+1), ...` will walk down a diagonal of the matrix, starting at element i.

⟨*coxmeFull-init-2*⟩=
```
gtemp <- as.factor(G[[1]])[,drop=TRUE] #drop unused levels
nlevel <- length(levels(gtemp))
gfrac <- table(gtemp)/ length(gtemp)
if (nlevel >= sparse[1] && any(gfrac <= sparse[2])) {
    indx <- order((gfrac> sparse[2]), 1:nlevel)  #False then True for order
    nsparse <- sum(gfrac <= sparse[2])
    if (nsparse== nlevel) vmat<- bdsI(nsparse)
    else {
        k <- nlevel - nsparse  #number of non-sparse levels
```

```
        rmat <- matrix(0., nrow=nlevel, ncol=k)
        rmat[seq(nsparse+1, by= nlevel+1, length=k)] <- 1.0
        vmat <- bdsmatrix(blocksize=rep(1,nsparse),
                          blocks= rep(1,nsparse), rmat=rmat)
        }
    }
 else {
    vmat <- diag(nlevel)
    indx <- 1:nlevel
    nsparse <- 0
    }
 imap <- matrix(match(as.numeric(gtemp), indx))
 levellist <- list((levels(gtemp))[indx])
```

Since the variance must be positive, iteration is done on the log value. The `levellist` and `gname` parts of the paramter list will be used by the wrapup function to create labels.

⟨*coxmeFull-init-2*⟩=
```
 varlist <- list(vmat)
 if (nvar==0)
    return(list(imap=imap, X=NULL,
                theta=(lapply(itheta, log))[!which.fixed],
                parms=list(varlist=varlist, theta=theta, levellist=levellist,
                           fixed=which.fixed, case=2, gname=gname)))
```

The generate function for this case is quite simple.

⟨*coxmeFull-generate-2*⟩=
```
 if (parms$case==2) return(theta*parms$varlist[[1]])
```

Matching any user input for either the `vfixed` or `vinit` arguments (which show up here as `fixed] and [[vinit`) for cases 2 and 3 can be done by a common bit of code since the names have to match up precisely with the grouping variables. For reasons discussed below we order the parameters from the last grouping variable to the first.

⟨*initialize-inits*⟩=
```
 gname <-  names(G)
 ntheta <- length(gname)
 itheta <- vector('list', length=ntheta)
 for (i in 1:ntheta) itheta[[i]] <- control$varinit
 if (ntheta >1) {
    for (i in 2:ntheta) gname[i] <- paste(gname[i-1], gname[i], sep='/')
    gname <- rev(gname)
    }
 names(itheta) <- gname

 if (length(vinit) >0) {
```

```
    temp <- initmatch(gname, vinit)
    if (any(temp==0))
        return(list(error=paste('Element', which(temp==0),
                                'of initial values not matched')))
    else itheta[temp] <- vinit
    if (any(unlist(vinit) <=0))
        return(list(error='Invalid initial value'))
    }

theta <- rep(0, ntheta)   # the filler value does not matter
which.fixed <- rep(FALSE, ntheta)
if (length(fixed)>0) {
    temp <- initmatch(gname, fixed)
    if (any(temp==0))
        return(list(error=paste('Element', which(temp==0),
                                 'of variance values not matched')))
    else theta[temp] <- unlist(fixed)
    which.fixed[temp] <- TRUE
    }
```

The third case is an intercept with nested grouping variables. We first expand out the second variable using the `expand.nested` routine; for a term such as `(1 | school/teacher)` we need to relabel the `teacher` variable so that teacher 1 in school A is different than teacher 1 in school B. This will lead to a stucture with g_1 levels for the first variable $g_1 * g_2$ levels for the second, and so on. This leads to two columns in `imap`, one for each variable, corresponding to the following structure.

$$\begin{aligned} b_i &\sim N(0, \sigma_1^2 I) \\ c_{ij} &\sim N(0, \sigma_2^2 I) \end{aligned}$$

Sparseness is applied to the *last* variable in the nesting, since it has the largest number of levels. This is done by reversing the parameters. Note that the `expand.nested` routine has already remomved any unused levels.

⟨*coxmeFull-init-3a*⟩=
```
G <- rev(expand.nested(G))  #the last shall be first
imap <- matrix(0L, nrow=nrow(G), ncol=ngroup)
imap[,1] <- as.numeric(G[,1])
for (i in 2:ngroup)
    imap[,i] <- as.numeric(G[,i]) + max(imap[,i-1])
levellist <- lapply(G, levels)
nlevel <- sapply(levellist, length)

# Sparsity?
gtemp <- G[,1]
gfrac <- table(gtemp)/ length(gtemp)
```

```
 if (nlevel[1] > sparse[1] && any(gfrac <= sparse[2])) {
     indx <- order((gfrac> sparse[2]), 1:nlevel[1])
     nsparse <- sum(gfrac <= sparse[2])

     imap[,1] <- match(as.integer(gtemp), indx)
     levellist[[1]] <- (levellist[[1]])[indx]
     }
 else  nsparse <- 0  #a single sparse element is the same as dense
```

The final variance matrix is diagonal with with rep(theta, nlevel) down the diagonal. Create a set of ngroup matrices all the same shape, each with 1's the right place on the diagonal, so that their sum is what we need.

⟨*coxmeFull-init-3a*⟩=
```
 if (nsparse==0) tmat <- diag(sum(nlevel))
 else tmat <- bdsmatrix(blocksize=rep(1L, nsparse), blocks=rep(1., nsparse),
                   rmat=matrix(0., nrow=sum(nlevel), ncol=sum(nlevel)-nsparse))
 varlist <- vector('list', ngroup)
 for (i in 1:ngroup) {
     temp <- rep(0., nrow(tmat))
     temp[unique(imap[,i])] <- 1.0
     temp2 <- tmat
     diag(temp2) <- temp
     varlist[[i]] <- temp2
     }

 if (nvar==0)
     return(list(imap=imap, X=NULL,
                 theta=(lapply(itheta, log))[!which.fixed],
                 parms=list(nlevel=nlevel, varlist=varlist, gname=names(G),
                            fixed=which.fixed, levellist=levellist,
                            theta=theta, case=3, collapse=FALSE)))
```

Now all that the generate function needs to do is to add the weighted matrices. We want the generate functions to be simple, since they are executed hundreds of times.

⟨*coxmeFull-generate-3*⟩=
```
 if (parms$case==3) {
     temp <- theta[1]* parms$varlist[[1]]
     for (i in 2:length(parms$varlist))
         temp <- temp + theta[i]*parms$varlist[[i]]
     return(temp)
     }
```

Although the above is a simple approach, we have found the program is often more stable using an alternate representation. I hypothesise that this is due to a smaller number of nuisance variables.

Update: I'm leaving the code in (it does work), but have since realized that it was all based on a misunderstanding. When computed correctly the collapse=TRUE and collapse=FALSE lead to identical iteration paths. The observation that led to doing a collapse option was in the prior code, with a large number of groups, and I was unwittingly using different patterns of sparsity.

Consider again a 2 level nesting b/c and let

$$\begin{aligned} d_{ij} &= b_i + c_{ij} \\ d &\sim N(0, A) \end{aligned}$$

Then A is a block diagonal array with one block for each level of the primary grouping variable$b$, and

$$\begin{aligned} A_{ii} &= \sigma_1^2 + \sigma_2^2 \\ A_{ij} &= \sigma_1^2 \end{aligned}$$

for i and j in the same block, and 0 otherwise. The size of the first block is the number of unique levels of c that occur for the first level of b. We can treat the fit as a single random effect d, but with a more complex variance/covariance matrix between the terms.

Sparsity is more complex– we can only ignore elements that are both not part of the penalty and are ok combinations for the Cox model hessian. The first of these is based on the block structure just above and depends on the first grouping variable. The Cox sparsity is based on d, covariances can be ignored for any pair of levels in which both are sparse. The upshot is that we need to order the coefficients by block, with any sparse blocks (ones in which every d is sparse) first.

The `bdsBlock` function makes it fairly simple to create the blocks. At the end we assess sparseness, if ≤ 1 block counts as sparse we keep only one of them in the block portion, e.g., a dense matrix. For creating the matrices we need the number of unique coefficients = number of levels of the last element of the expanded G. So all this computation works on that unique subset.

⟨ *coxmeFull-init-3b* ⟩=

```
gtemp <- expand.nested(G)
n <- nrow(G)

#Sparse?
gfrac <- table(gtemp[,ngroup])/ n
nlevel <- length(levels(gtemp))
if (nlevel > sparse[1] && any(gfrac <= sparse[2])) {
        is.sparse <- (gfrac <= sparse[2])[as.numeric(gtemp[,ngroup])]
        block.sparse <- tapply(is.sparse, G[,1], all)
        nsparse
        }
else block.sparse <- 0
```

```
if (sum(block.sparse > 1)) { #sparse blocks exist, make them list first
    border <- order(!block.sparse, 1:nlevel)
    G[,1] <- factor(gtemp[,1], levels=levels(G[,1])[border])
    G <- expand.nested(G)
    }

G <- rev(expand.nested(G))
levellist <- lapply(G, levels)
nlevel <-  sapply(levellist, length)
imap <- matrix(as.numeric(G[,1]))

varlist <- vector('list', ngroup)
indx <- match(levellist[[1]], G[[1]])  #an ordered set of unique rows
for (i in 1:ngroup)
    varlist[[i]] <- bdsBlock(1:nlevel[1], G[indx,i])

if (sum(block.sparse) <=1) {#make them all ordinary matrices
    for (i in 1:ngroup) varlist[[i]] <- as.matrix(varlist[[i]])
    }
else {
    if (!all(block.sparse)) { # Only a part is sparse
        tsize <- sum(temp@blocksize[1:sum(block.sparse)]) # sparse coefs
        sparse <- 1:tsize  #sparse portion, remainder is dense
        smat <- (varlist[[ngroup]])[1:sparse, 1:sparse]
        varlist[[ngroup]]
        rmat <- matrix(0, sum(tsize), nlevel[1])
        rmat[seq(by=nrow(rmat)+1, to=length(rmat), length=ncol(rmat))] <- 1.0
        varlist[[ngroup]] <- bdsmatrix(blocksize=smat@blocksize,
                                       blocks=smat@block, rmat=rmat)
        }
    varlist <- bdsmatrix.reconcile(varlist)
    }

if (nvar==0) {
    return(list(imap=matrix(as.numeric(G[,1])), X=NULL,
                theta=lapply(itheta, log)[!which.fixed],
                parms=list(varlist=varlist, theta=theta,
                           fixed=which.fixed, gname=names(G),
                           levellist=levellist, case=3, collapse=TRUE)))
    }
```

The last case is the hardest; we have both grouping factors and covariates. To keep track of the coefficients I create two working variables, `imap` and `xmap`, the first corresponds to intercepts and the second to covariates. The `imap` matrix has n rows and one column for each level of grouping; for each subject it shows which intercept coefficients that subject participates in. The `xmap` matrix is similar; it shows the coefficient number(s) for each X variable. For a random

term `(1 + x1 + x2 + x3| g1/g2)` the retuned X matrix will have 6 columns since there are 2 sets of coefficients for every covariate, `xmap` contains the mapping to the set of coefficients corresponding to each column. At this time the underlying C code for `coxme` demads that `imap` point to the first block of coefficients and `xmap` to the next, i.e. that all intercept coefficients come first (this may eventually change). Thus `xmap` picks up where `imap` left off. Assume that I had 1 grouping variable with 9 levels and 2 covariates x_1, x_2. If we set the first column of `xmap` to `imap +9` and the second one to `imap +18`, then then coefficient pairs 1 and 10, 1 and 19, and 10 and 19 are correlated, but distinct ones within a column of `imat` or of `xmat` are not. This leads to the overall correlation matrix for the coefficients given below, where A is the 9 by 9 identiy matrix.

$$\begin{pmatrix} \sigma_1^2 A & \sigma_{12} A & \sigma_{13} A \\ \sigma_{12} A & \sigma_2^2 A & \sigma_{23} A \\ \sigma_{13} A & \sigma_{23} A & \sigma_3^2 A \end{pmatrix} \tag{1}$$

If there are multiple grouping variables they will come first: replace the upper left corner of 1 with the combined matrix $A(\theta)$ for the set, the other blocks are also $A(\theta)$ but using a different portion of the θ vector. For a grouping variable `g1/g2` with 2 levels for g_1 and 4 for the g_1/g_2 pairing we have

$$A(\sigma_1, \sigma_2) = \begin{pmatrix} \sigma_1^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & \sigma_1^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma_2^2 & 0 & 0 & 0 \\ 0 & 0 & 0\sigma_2^2 & 0 & 0 & \\ 0 & 0 & 0 & 0\sigma_2^2 & 0 & \\ 0 & 0 & 0 & 0 & 0 & \sigma_2^2 \end{pmatrix}$$

which is exactly the variance matrix we would have had for a `(1 | g1/g2)` term. The second block will be a function of the covariances between the two intercept terms and `x1`, $A(\sigma_{13}, \sigma_{23})$, and the upper right block the covariances between `x2` and the intercepts.

The set of paramters θ is most easily arranged in the following way: for each grouping variable we have an (nvar +1) by (nvar +1) set of variances/covariances, with the intercept as the first column. The θ vector has the lower triangle of this (in standard R matrix order) for the first grouping variable, then for the second, etc. The `tname` vector gives names to the elements, in order to allow a user to set selected values. For the random term `1 + x1 + x2 | g1/g2` the names would be `g1, x1:g1, x2:g1, x1/g1, x1:x2/g1, x2/g1`. (I don't particularly like these; if you can think of a better naming scheme let me know.) The default values for θ are 0 and .3 for the correlations and $(.02, .1, .3, .9)^2$ for the variances. For computation they are transformed with variances=e^θ and correlations=$(e^\theta - 1)/(e^\theta + 1)$.

Having worked all this out now throw one more complication into the mix. Again look at the term `(1 + x1 + x2 | g1)`, with the 6 parameters $(\sigma_1^2, \sigma_2^2, \sigma_3^2, \sigma_{12}, \sigma_{13}, \sigma_{23})$ which are variances of the intercept, `x1`, and `x2` coefficients along with their covariances. If `x1` and `x2` are from the same term, e.g., they are the 0/1 dummy variables for two levels of a factor, then we assume that $\sigma_2^2 = \sigma_3^2$, $\sigma_{12} = \sigma_{13}$ and $\sigma_{23} = 0$; from a parameter count view they act like a single variable. This makes no change at all in the number of coefficients b or in the structure of their covariance matrix in equation 1. But we now need to have a "real" θ vector containing just 3 parameters, an expanded one `etheta` containing all 6 terms, and a mapping between them.

The last point to mention before actual code is whether the Xcovariates should be centered or scaled. There is a very good reason for doing this in the coxph code, since βx and $\beta(x+10000000)$

have the same log-likelihood for any value of β but the latter one can cause the exp function to overflow while doing the calculations. It this can happen, for instance when `x` is a Date object. I'm so used to this that I originally built the idea into this code before realizing it causes a problem: in a `(1+ x1 | g1)` model for instance subtracting a constant from `x1` changes the variance estimate for the intercept term of `g1`. (The same is true for linear mixed models). It does lead to the same log-likelihood and thus correct tests, but the change in printed output should be avoided. Scaling the colums of X causes problems for the `refine.n` code in the main program; it saw the rescaled X matrix but not the rescaled coefficients. Currently we do rescale the default starting estimates, thus if a user replaces X with $2X$ the code will follow the same iteration path.

⟨*coxmeFull-init-4*⟩=
```
 xvar  <- apply(X,2,var)
 cordefault <- control$corinit
 itheta <- list()
 is.variance <- NULL
 if (intercept) {
     itheta <- c(itheta, list(control$varinit))
     for (i in 1:ncol(X)) itheta <- c(itheta, list(control$corinit)) #correlations
     is.variance <- c(TRUE, rep(FALSE, ncol(X)))
     }
 for (i in 1:ncol(X)) {
     itheta <- c(itheta, list(control$varinit/xvar))  #variance
     if (i < ncol(X)) {
         for (j in (i+1):ncol(X))  itheta <- c(itheta, list(cordefault))
         is.variance <- c(is.variance, TRUE, rep(FALSE, ncol(X)-i))
         }
     else is.variance <- c(is.variance, TRUE)
     }
 itheta <- rep(itheta, ngroup)
 is.variance <- rep(is.variance, ngroup)

 xname <- dimnames(X)[[2]]
 name.temp <- outer(xname, xname, paste, sep=":")
 diag(name.temp) <- xname
 name.temp <- name.temp[row(name.temp) >= col(name.temp)]
 tname <- ""
 gname <- names(G)
 for (i in 1:ngroup) {
     if (intercept)
         tname <- c(tname, gname[i], paste(xname, gname[i], sep=':'),
                    paste(name.temp, gname[i], sep='/'))
     else tname <- paste(name.temp, gname[i], sep='/')
     }

 # Now replace selected values with the user's input
```

```
if (length(vinit) > 0) {
    temp <- initmatch(tname, vinit)
    if (any(temp==0))
        return(list(error=paste('Element(s)', which(temp==0),
                                'of initial values not matched')))
    else itheta[temp] <- unlist(vinit)
    }

which.fixed <- rep(FALSE, length(itheta))
if (length(fixed) > 0) {
    temp <- initmatch(tname, fixed)
    if (any(temp==0))
      return(list(error=paste('Element(s)', which(temp==0),
                              'of fixed variance values not matched')))
    else itheta[temp] <- unlist(fixed)
    which.fixed[temp] <- TRUE
    }

# Check for legality of the values
tmat <- diag(nvar+ intercept)  #dummy variance/cov matrix
tmat <- tmat[row(tmat) >= col(tmat)]
vindx <- which(tmat==1)  #indices of the variance terms within each group
cindx <- which(tmat==0)  #indices of the correlations

if (any(unlist(itheta[is.variance]) <=0))
        return(list(error="Variances must be >0"))
if (any(unlist(itheta[!is.variance]) <=-1) || any(unlist(itheta[!is.variance]) >=1))
        return(list(error="Correlations must be between 0 and 1"))
```

If there is no intercept in the random effects formula then `xmap` should start at 1, otherwise the X coefficients come after the intercepts.

⟨*coxmeFull-init-4*⟩=

```
xnew <- matrix(0., nrow=nrow(X), ncol=nvar*ncol(imap))
xmap <- matrix(0L, nrow=nrow(X), ncol=ncol(X)*ncol(imap))
xoffset <- (intercept)* max(imap)
k <- 1
for (j in 1:nvar) {
    for (i in 1:ncol(imap)) {
        xnew[,k] <- X[,j]
        xmap[,k] <- imap[,i] + xoffset
        k <- k+1
        xoffset <- xoffset + max(imap)
        }
    }

# Transform correlations to (1+rho)/(1-rho) scale, which is used for the saved
```

```
#  parameters, and make a copy.  The initial parameters are then log transformed
itheta[!is.variance] <- lapply(itheta[!is.variance], function(rho) (1+rho)/(1-rho))
theta <- sapply(itheta, function(x) x[1])

itheta <- lapply(itheta, log)

if (intercept)
    list(theta=itheta[!which.fixed], imap=imap, X=xnew, xmap=xmap,
             parms=list(theta=theta, fixed=which.fixed,
                        nlevel=nlevel, levellist=levellist,
                        nvar=nvar, gname=names(G), varlist=varlist,
                        xname=dimnames(X)[[2]], intercept=intercept,
                        xname=dimnames(X)[[2]], case=4, collapse=collapse))
else list(theta=itheta[!which.fixed], imap=NULL, X=xnew, xmap=xmap,
             parms=list(theta=theta, fixed=which.fixed,
                        nlevel=nlevel, levellist=levellist,
                        nvar=nvar, gname=names(G), varlist=varlist,
                        xname=dimnames(X)[[2]], intercept=intercept,
                        xname=dimnames(X)[[2]], case=4, collapse=collapse))
```

The generation routine needs to create the full variance-covariance matrix of the parameters, which is fortunately of a structured form. Looking at the matrix 1, the diagonal blocks are first the variance-covariance of the intercepts, then that of the regression coefficients for the first covariate, the second, etc. The top row contains covariances between the intercept and the covariates.

All of these matrices *have exactly the same form*! This means that we keep adding up the same prototype matrices from the varlist, but using different coefficients. If there are multiple grouping variables the θ vector consists of blocks, one per grouping variable; all are processed at once. First all the intercepts at once, then the interceps* first covariate slope term, intercepts * second covariate, etc.

⟨*coxmeFull-generate-4*⟩=

```
 if (parms$case==4) {
     ngroup <- length(parms$nlevel)
     n1 <- sum(parms$nlevel)          #number of intercept coefs
     nvar <- parms$nvar               #number of covariates
     n2 <-   n1*nvar                  #number of slope coefs
     theta.per.group <- length(theta)/ngroup
     tindx <- seq(1, by=theta.per.group, length=ngroup)

     addup <- function(theta, p=parms) {
         tmat <- theta[1] * p$varlist[[1]]
         if (length(theta) >1) {
             for (i in 2:length(theta)) tmat <- tmat + theta[i]*p$varlist[[i]]
             }
         tmat
         }
```

```
if (parms$intercept) {
    # upper left corner (has no covarinaces)
    ivar <- theta[tindx]  #variances of the intercepts
    corner <- addup(ivar)
    if (inherits(corner, 'bdsmatrix')) {
        nsparse <- sum(corner@blocksize)
        rmat <- matrix(0., nrow=n1+n2, ncol=n1+n2 - nsparse)
        if (nsparse < n1) rmat[1:n1, 1:(n1-nsparse)] <- corner@rmat
        }
    else {
        nsparse <- 0
        rmat <- matrix(0., n1+n2, n1+n2)
        rmat[1:n1, 1:n1] <- corner
        }

    # Covariances with the intercept
    for (i in 1:nvar) {
        xvar <- theta[i+nvar+tindx]  #variance of the slope
        xcor <- (theta[i+tindx]-1)/(theta[i+tindx]+1)  # correlation
        icov <- xcor * sqrt(xvar * ivar)               # covariance
        rmat[1:n1, 1:n1 +i*n1 -nsparse] <- as.matrix(addup(icov))
        }
    irow <- n1
    icol <- n1 - nsparse
    theta <- theta[-(1:(1+nvar))]  #these thetas are 'used up'
    }
else {
    irow <- icol <- 0
    rmat <- matrix(0., n2,n2)
    }

# covariates
offset1 <- 0
for (i in 1:nvar) {
    xvar <- theta[offset1 + tindx]  #variance of the slope
    rmat[1:n1 + irow, 1:n1 + icol] <- as.matrix(addup(xvar))
    # covariate-covariate
    if (i<nvar) {
        offset2 <- offset1 + 1 + nvar-1
        for (j in (i+1):nvar) {
            icol <- irow + n1
            zvar <- theta[offset2 + tindx]-1
            zcor <- (theta[j+offset2+tindx] -1)/(theta[j+offset2 +tindx]+1)
            zcov <- sqrt(xvar*zvar) * zcor
            rmat[1:n1+ irow, 1:n1 + icol] <- as.matrix(addup(zcov))
```

```
                offset2 <- offset2 + 1 + nvar -j
                }
            offset1 <- offset1 + 1 + nvar- i
            icol <- irow <- irow+n1
            }
        }

    if (parms$intercept && inherits(corner, 'bdsmatrix'))
        bdsmatrix(blocksize=corner@blocksize, blocks=corner@blocks, rmat=rmat)
    else bdsmatrix(blocksize=integer(0), blocks=numeric(0), rmat=rmat)
    }
```

The wrapup function transforms theta back, adds names, and formats the vector of random coefficients b. For cases 1 and 2 adding names is almost all we need to do.

⟨*coxmeFull-wrapup*⟩=

```
 wrapup <- function(theta, b, parms) {
    newtheta <- parms$theta
    if (length(theta)) newtheta[!parms$fixed] <- exp(theta)

    if (parms$case==1) {
        theta <- list(c'(Shrinkage)' = newtheta[1]))
        names(theta) <- '1'
        names(b) <- parms$xname
        return(list(theta =theta, b=list('1'=b)))
        }

    if (parms$case==2) {
        names(newtheta) <- 'Intercept'
        names(b) <- parms$levellist[[1]]
        theta <- list(newtheta)
        names(theta) <- parms$gname
        b <- list(b)
        names(b) <- parms$gname
        return(list(theta=theta, b=b))
        }
```

For case 3, we need to distinguish between `collapse` equal true or false. For the former, there will be `ngroup` random parameters but only a single vector of coefficients b. For the latter there will be one set of b coefficients for each level of the random effect.

⟨*coxmeFull-wrapup*⟩=

```
    if (parms$case==3) {
        ngroup <- length(parms$levellist)
        theta <- vector('list', ngroup)
        names(theta) <- parms$gname
        for (i in 1:ngroup)
```

```
        theta[[parms$gname[i]]] <- c('(Intercept)'=newtheta[i])

    if (parms$collapse) {
        names(b) <- parms$levellist[[1]]
        random <- list(b)
        names(random) <- parms$gname[[1]]
        }
    else {
        names(b) <- unlist(parms$levellist)
        random <- split(b, rep(1:ngroup, parms$nlevel))
        names(random) <- parms$gname
        }
    return(list(theta=theta, b=random))
    }
```

The last case is of course the most complicated, it has both covariates and groupings. For a complicated random effect `(1+ age |institution/sex)` it should return a two element list for `theta` with names 'institution' and 'institution/sex', each of which contains a 2×2 matrix with variances on the diagonal and correlations off the diagonal. (We are echoing the desired form for the printout). In the case that intercept is false and nvar=1, e.g. the formula `(age | institution/sex)`, each element of the list is a one-element vector rather than a matrix.

The random effect will also be a list of two elments, each a matrix with 2 columns containing the coefficients for the intercept and age. It may be a matrix of one column.

⟨*coxmeFull-wrapup*⟩=

```
 if (parms$case==4) {
        intercept <- parms$intercept
        ngroup <- length(parms$nlevel)
        nvar <- parms$nvar

        # Deal with b
        random <- split(b, rep(rep(1:ngroup, parms$nlevel), intercept +nvar))
        names(random) <- parms$gname
        if (intercept) {
            colname <- c("Intercept", parms$xname)
            }
        else {
            colname <- parms$xname
            }

        for (i in 1:ngroup) {
            temp<- matrix(random[[i]], ncol=length(colname))
            random[[i]] <- temp
            dimnames(random[[i]]) <- list(parms$levellist[[i]], colname)
            }

        # Deal with theta
```

```
        tfun <- function(x, n= 1 + nvar) {
            tmat <- matrix(0., n, n)
            tmat[row(tmat) >= col(tmat)] <- x
            offdiag <- row(tmat) > col(tmat)
            tmat[offdiag] <- (tmat[offdiag]-1)/(tmat[offdiag]+1)
            dimnames(tmat) <- list(colname, colname)
            tmat + t(tmat) - diag(diag(tmat))
            }
        parms.per.group <- length(newtheta)/ngroup
        if (parms.per.group==1) { #nvar=1, intercept=F case
            theta <- as.list(newtheta)
            theta <- lapply(theta, function(x) {names(x)<- parms$xname; x})
            names(theta) <- parms$gname
            }
        else {
            theta <- vector('list', ngroup)
            names(theta) <- parms$gname
            for (i in 1:ngroup)
                theta[[i]] <- tfun(newtheta[1:parms.per.group +
                                            parms.per.group*(i-1)])
            }
        return(list(theta=theta, b=random))
        }
    }
```

4.4 coxmeMlist

In a mixed-effects model the random effects b are assumed to follow a Gaussian distribution

$$b \sim N(0, \Sigma)$$

In all the random effects modeling programs that I am aware of, the user specifies the structure of Σ and the program constructs the actual matrix. For instance, 'independent', 'compound symmetry', or 'autoregressive'. This basic approach does not work for genetic studies, since the correlation is based on family structure and cannot be inferred from a simple keyword. The `coxmeMlist` variance specification accepts a list of fixed matrices A_1, A_2, ... and fits the variance structure $\Sigma = \sigma_1^2 A_1 + \sigma_2^2 A_2 + \ldots$. The individual matrices are often in a block-diagonal sparse representation due to size. (The motivating study for this structure had 26050 subjects with a random intercept per subject, so that A was 26050 by 26050.)

The matrices must have dimnames that match the levels of the grouping variable. Much of the initialization work is to verify this, remove unneeded columns of the matrices (if for instance a subject has been dropped due to missing values), and reorder the grouping variable to match the resulting matrix. We can't reorder sparse matrices willy-nilly without (potentially) creating disastrous consequences wrt losing sparsity in the matrix decompostion. So unlike most fitting routines that will create dummy coeffcients that are in the order of the levels of the grouping variable, we keep random effects in the order that they appear in the matrix A. The `imat` vector

contains the "new" covariate for each subject, e.g., the first person is in group 10, the next in group 2, etc. If there are multiple matrices, then the bsdmatrix.reconcile routine spends most of it's energy deciding if they can be put into a common row/column ordering. (If designing this over, I'd likely have the routine just check "are they the same" and give an error message otherwise.)

Because of this complexity, I only allow terms of the form `(1|g)` or `(x|1)` (x could be a matrix or list of variables, g could be nested) when a user-specified matrix is involved.

Three checks on the matrices are commonly added.

1. A solution with $A^* = A/2$ and $\sigma^* = \sigma\sqrt{2}$ is of course equivalent to one with A and σ. For uniqueness, the matrices A_1, A_2 etc are rescaled to have a diagonal of 1. Kinship matrices in particular often have a diagonal of 1/2.

2. The individual A matrices are checked to verify that each is positive definite. If they are not this is most often reflects an error in forming them. With the extension of the package to more general Matrix objects this parameter's default has been reset to FALSE, as it can cause a large amount of confusion when it is applied to other cases, e.g., a smoothness penalty.

3. The parameters σ are constrained to be > 0.

I have had analyses where each of these had to be relaxed.

```
⟨coxmeMlist⟩=
 coxmeMlist <- function(varlist, rescale=FALSE, pdcheck=TRUE,  positive=TRUE) {
     # Because of environments, the init function will inherit the
     #  four variables below
     varlist <- varlist
     rescale <- rescale
     pdcheck <- pdcheck
     if (!is.logical(positive)) stop("Invalid value for postive argument")
     positive <- positive
     ⟨coxmeMlist-init⟩
     ⟨coxmeMlist-generate⟩
     ⟨coxmeMlist-wrapup⟩
     out <- list(initialize=initialize, generate=generate, wrapup=wrapup)
     class(out) <- 'coxmevar'
     out
     }
```

The `initialize` routine needs to match each row/column of the variance matrix or matrices that have been given to the appropriate element of the random coefficient b; this is done using the dimnames. (The matrices must be square). Most of the real work is done by `bdsmatrix.reconcile`. Given a list of variance matrices and a list of target dimnames, it returns a list where all the matrices have the same row/col order, the dimnames of which will be the order of the coefficients b. It also drops any unused rows or cols from the matrices.

The `bdsmatrix.reconcile` routine expects dimnames on all the matrices. If none of the matrices are given a dimname, we add them before calling the routine — but only if they are

exactly the right dimension. This allows a user to give an unnamed matrix that is just exactly the right length.

⟨*coxmeMlist-init*⟩=
```
 initialize <- function(vinit, fixed, intercept, G, X, control) {
     ngroup <- min(length(G), ncol(G))
     nvar   <- min(length(X), ncol(X))  # a NULL or a nx0 matrix yields 0
     if (ngroup >0 & nvar >0)
         return(list(error="Mlist cannot have both covariates and grouping"))
     if (!is.list(varlist)) varlist <- list(varlist)  # a naked matrix
     noname <- all(sapply(varlist, function(x) is.null(dimnames(x)) ||
                      (is.null(dimnames(x)[[1]]) & is.null(dimnames(x)[[2]]))))
     namefun <- function(x, names) {
         if (all(dim(x)== rep(length(names),2)))
             dimnames(x) <- list(names, names)
         x
         }
     if (ngroup >0) {
         n <- nrow(G)
         G <- expand.nested(G)
         groups <- G[[ngroup]]  #drop all but the last
         if (noname) varlist <- lapply(varlist, namefun, levels(groups))
         if (any(sapply(varlist, function(x) inherits(x, "Matrix"))))
             varlist <- lapply(varlist, function(x) as(x, "bdsmatrix"))
         tlist <- bdsmatrix.reconcile(varlist, levels(groups))
         bname <-  dimnames(tlist[[1]])[[1]]
         imap <- matrix(match(groups, bname))
         xmap <- NULL
         rname <- names(G)[[ngroup]]
         }
     else {
         n <- nrow(X)
         bname <- dimnames(X)[[2]]
         if (noname) varlist <- lapply(varlist, namefun, bname)
         tlist <- bdsmatrix.reconcile(varlist, bname)
         # sparse matrices (bdsmatrix or Matrix) are illegal, for now,
         #   for covariates
         tlist <- lapply(tlist, as.matrix)
         xmap <- match(dimnames(X)[[2]], bname)
         xmap <- matrix(rep(xmap, n), nrow=n, byrow=T)
         imap <- NULL
         rname <- "(Shrink)"
         }

     ⟨Mlist-initial-value⟩
     ⟨Mlist-matrix-checks⟩
```

```
    # itheta is a list with vectors of initial values
    # theta is a vector, and only the fixed values need to be correct (the others
    #  are replaced by the parent routine).  All fixed "inits" are of length 1.
    theta <- sapply(itheta, function(x) x[1])
    list(theta=itheta[!which.fixed], imap=imap, X=X, xmap=xmap,
         parms=list(varlist=tlist, theta=theta, fixed=which.fixed,
                    bname=bname, rname=rname, positive=positive,
                    vname=names(varlist)))
    }
```

Processing initial values is very simple: the number of coefficients is equal to the number of matrices in the varlist. Names are ignored, zeros are treated as "missing". In some genetics problems having all the variances equal leads to singularity, so we fudge the default initial values.

⟨*Mlist-initial-value*⟩=

```
ntheta <- length(varlist)
fudge <- seq(1, 1.5, length=ntheta)
itheta <- vector('list', ntheta)
for (i in 1:ntheta) itheta[[i]] <- control$varinit * fudge[i]

if (length(vinit) >0) {
    if (length(vinit) != ntheta)
        return(list(error="Wrong length for initial values"))
    indx <- !is.na(vinit) & vinit !=0  #which to use
    if (any(indx)) itheta[indx] <- vinit[indx]
    }

which.fixed <- rep(FALSE, ntheta)
if (length(fixed) >0) {
    if (length(fixed) != ntheta)
        return(list(error="Wrong length for fixed values"))
    indx <- !is.na(fixed) & fixed !=0  #which to use
    if (any(indx)) {
        itheta[indx] <- fixed[indx]
        which.fixed[indx] <- TRUE
        }
    }

if (length(positive)==1) positive <- rep(positive, ntheta)
if (length(positive) != ntheta)
    return(list(error="Wrong length for positive parameter"))
if (any(unlist(itheta[positive]) <=0))
    return(list(error="Invalid initial value, must be positive"))
itheta[positive] <- lapply(itheta[positive], log)
```

Check the matrices for validity. We use non-negative definite (NND) rather than positive definite because identical twins generate a NND kinship matrix.

⟨*Mlist-matrix-checks*⟩=
```
 for (j in 1:ntheta) {
     kmat <- tlist[[j]]
     if (rescale) {
         temp <- diag(kmat)
         if (any(temp==0))
             return(list(error="Diagonal of a variance matrix is zero"))
 #        if (any(temp != temp[1]))
 #            warning("Diagonal of variance matrix is not constant")
         if (max(temp) !=1) {
             kmat <- kmat/max(temp)
             tlist[[j]] <- kmat
             }
         }
     if (pdcheck) {
         temp <- gchol(kmat)
         if (any(diag(temp) < 0))
             return(list(error="A variance matrix is not non-negative definite"))
         }
     }
```

The generate function is a simple sum.

⟨*coxmeMlist-generate*⟩=
```
  generate <- function(newtheta, parms) {
      theta <- parms$theta
      theta[!parms$fixed] <- newtheta
      if (any(parms$positive)) theta[parms$positive] <-
           exp(pmax(-36, pmin(36, theta[parms$positive])))

      varmat <- parms$varlist[[1]] * theta[1]
      if (length(theta) >1) {
          for (i in 2:length(theta)) {
              varmat <- varmat + theta[i]*parms$varlist[[i]]
              }
          }
      varmat
      }
```

Wrapup is also simple. The thetas are named Vmat.1, Vmat.2, etc; or using the names found on the orginal varlist (if any).

⟨*coxmeMlist-wrapup*⟩=
```
 wrapup <- function(newtheta, b, parms) {
         theta <- parms$theta
         theta[!parms$fixed] <- newtheta
         theta[parms$positive] <- exp(theta[parms$positive])
```

```
defaultname <- paste("Vmat", 1:length(theta), sep=".")
vname <- parms$vname
if (length(vname)==0) vname <- defaultname
else if (any(vname==''')){
    indx <- which(vname==''')
    vname[indx] <- defaultname[indx]
    }
names(theta) <- vname
theta <- list(theta)
names(theta) <- parms$rname
names(b) <- parms$bname
b <- list(b)
names(b) <- parms$rname
list(theta=theta, b=b)
}
```

5 Fitting

Consider the basic model

$$\begin{aligned} \lambda(t) &= \lambda_0(t) e^{X\beta + Zb} \\ b &\sim N(0, \Sigma(\theta)) \end{aligned}$$

There are two sets of parameters. The first is the set of regression coefficient β and b, the second is the vector θ that determines the variance structure. The basic structure of the iteration is

- an outer iteration process for θ which uses the standard S routine `optim`
- for any given realization of θ a computation of the optimal values for β and b
  - S code is used to create the penalty matrix $\Sigma(\theta)$
  - C code solves for the regression coefficients, given Σ.

The overall outline of the routine is

```
⟨coxme.fit⟩=
 ⟨coxme-setup⟩
     ⟨null-fit⟩
     ⟨define-penalty⟩
     ⟨coxfit6a-call⟩
     ⟨coxme-fit⟩
     ⟨coxme-finish⟩
 }
```

5.1 Penalty matrix

For the C code, the variance matrices of the individual random effects are glued together into one large bdsmatrix object Σ, `kmat` in the code. When `inverse=TRUE` (the default) the inverse matrix $P = \Sigma^{-1}$ or `ikmat` is the penalty matrix of the computation, and is what is actually passed to C. When it is false the user is working directly with penalty matrices. (The first large use of this code was for family correlation, where Σ is based on the *kinship* matrix. The variable names `kmat` $= \Sigma$, `ikmat` for the inverse and `kfun` for the calculation arise from this legacy.) In order to make use of sparseness, the columns of `kmat` are expected to be in the following order

1. Random intercepts that are subject to sparse computation. Only one random term is allowed to use sparse representation, i.e., the first term in the model formula that has an intercept. We have reordered the random terms, if necessary, to make it first in the list.
2. The remaining random intercepts
3. Other random coefficients (slopes)

The overall coefficient vector has the random effects b followed by the fixed effects β, with b in the same order as the penalty matrix.

The key code chunk below creates kmat given the parameter vector θ (`theta` for the non-mathematics types) and the variance list information. Each of the `generate` functions creates either an ordinary matrix or one represented in *b*lock *d*iagonal *symmetric* form, which consists of a block diagonal portion in the upper left bounded by a dense portion on the right. (A bdsmatrix with only one diagonal block is dense, one with many blocks will be sparse.) The C code expects a single bdsmatrix, so any term after the first is is added to the dense portion of the first matrix. It also expects the bdsmatrix to have at least one "block", and that block must involve no more than the first column of F. If the first term is a simple matrix, we just split off its first element.

⟨*define-penalty*⟩=
```
 kfun <- function(theta, varlist, vparm, ntheta, ncoef) {
     nrandom <- length(varlist)
     sindex <- rep(1:nrandom, ntheta) #which thetas to which terms

     tmat <- varlist[[1]]$generate(theta[sindex==1], vparm[[1]])
     dd <- dim(tmat)
     if (length(dd) !=2 || any(dd != rep(ncoef[1,1]+ncoef[1,2], 2)))
         stop("Incorrect dimensions for generated penalty matrix, term 1")
     if (!inherits(tmat, 'bdsmatrix'))
         tmat <- bdsmatrix(blocksize=integer(0), blocks=numeric(0), rmat=tmat)
```

If there is only a single random term, then our work is done. Otherwise we have some nit-picky bookkeeping. Not particularly hard but a nuisance. Say for example that there are 3 terms with the following structure

	sparse intercept	non-sparse intercept	covariate
term 1	60	2	64
term 2	0	5	0
term 3	0	8	16

Here "sparse" means precisely the block-diagonal portion of the returned variance matrix. This corresponds to `~ (1+x | g1) + (1|g2) + (1+ z1 + z2 | g3)` where `g1` has two common and 60 uncommon levels, `g2` has 5 levels and `g3` has 8. (A complicated random effects model I admit.) The first variance matrix is required to be of a `bsdmatrix` form with an `rmat` slot of dimension 126×66, call this T. The variance structure for the other two terms, call them U and V, can be of any matrix type. The final bdsmatrix will have the block-diagonal portions for the first 60 elements and an overall right-hand side matrix of the form

$$R = \begin{pmatrix} T[1-62, 1-2] & 0 & 0 & T[1-62, 3-66] & 0 \\ 0 & U & 0 & 0 & 0 \\ 0 & 0 & V[1-8, 1-8] & 0 & V[1-8, 9-24] \\ 0 & 0 & 0 & T[63-128, 3-66] & 0 \\ 0 & 0 & 0 & 0 & V[9-24, 9-24] \end{pmatrix}$$

First we set up the total number of rows and columns of R, then march through the matrix. We need to first process the non-sparse rows of the first variance matrix `tmat`; if that contains a substantial number of sparse columns then it is important to subset *before* creating a regular matrix from the remainder; the construction `(as.matrix(tmat))[k,k]` would create a temporary matrix of possibly vast proportions. At this point in time all of the intercepts map before any covariates so we can keep separate indices for the intercept-rows-so-far `indx1` and covariate-rows-so-far `indx2`, with the second one starting after the end of all the intercepts. What we are doing is in essence a diagonal bind of matrices, pasting blocks down the diagonal, but S has no `dbind` function.

⟨*define-penalty*⟩=
```
    if (nrandom ==1) return(tmat)

    # Need to build up the matrix by pasting up a composite R
    nsparse <- sum(tmat@blocksize)
    nrow.R <- sum(ncoef)
    ncol.R <- nrow.R - nsparse
    R <- matrix(0., nrow.R, ncol.R)
    indx1 <- 0                  #current column offset wrt intercepts
    indx2 <- sum(ncoef[,1]) -nsparse #current col offset wrt filling in slopes

    if (ncol(tmat) > nsparse) { #first matrix has an rmat component
        if (ncoef[1,1] > nsparse) { #intercept contribution to rmat
            irow <- 1:ncoef[1,1]  #rows for intercepts
            j <- ncoef[1,1] - nsparse   #number of dense intercept columns
            R[irow, 1:j] <- tmat@rmat[irow,1:j]
            indx1 <- j  #number of intercept processed so far

            if (ncoef[1,2] >0) {
                # T[1-62, 3-66] of the example above
                k <- 1:ncoef[1,2]
                R[irow, k+indx2-nsparse] <- tmat@rmat[irow, k+j]
            }
```

```
            }
        else j <- 0

        if (ncoef[1,2] >0) { #has a slope contribution to rmat
            # T[63-128, 3-66] of the example above
            k <- 1:ncoef[1,2]
            R[k+indx2 +nsparse, k+ indx2] <- tmat@rmat[k+indx1, j+k]
            indx2 <- indx2 + ncoef[1,2] #non intercetps so far
            }
        }

for (i in 2:nrandom) {
        temp <- as.matrix(varlist[[i]]$generate(theta[sindex==i], vparm[[i]]))
        if (any(dim(temp) != rep(ncoef[i,1]+ncoef[i,2], 2)))
            stop(paste("Invalid dimension for generated penalty matrix, term",
                       i))

        if (ncoef[i,1] >0)  { # intercept contribution
            #U or V [1-8, 1-8] in the example above
            j <- ncoef[i,1]
            R[indx1 +1:j + nsparse, indx1 +1:j] <- temp[1:j,1:j]

            if (ncoef[i,2] >0) {
                # V[1-8, 9-24] in the example
                k <- 1:ncoef[i,2]
                R[indx1+ 1:j + nsparse, indx2 +k] <- temp[1:j, k+ j]
                # V[9-24, 9-24]
                R[indx2+k +nsparse, indx2 +k] <- temp[k+j, k+j]
                }
            }
        else if (ncoef[i,2]>0) {
            k <- 1:ncoef[i,2]
            R[indx2+k +nsparse, indx2+k] <- temp
            }
        indx1 <- indx1 + ncoef[i,1]
        indx2 <- indx2 + ncoef[i,2]
        }
    bdsmatrix(blocksize=tmat@blocksize, blocks=tmat@blocks, rmat=R)
    }
```

5.2 C routines

The C-code underlying the computation is broken into 3 parts. This was done for memory efficiency; due to changes in R and S-Plus over time it may not as wise an idea as I once thought, this is an obvious area for future simplification.

The initial call passes in the data, which is then copied to local memory (using calloc, not

under control of S memory management) and saved. The parameters of the call are

n number of observations

nvar number of fixed covariates in X

y the matrix of survival times. It will have 2 columns for normal survival data and 3 columns for (start, stop) data

x the concatenated Z and X matrices

offset vector of offsets, usually 0

weights vector of case weights, usually 1

newstrat a vector that marks the end of each stratum. If for instance there were 4 strata with 100 observations in each, this vector would be c(100,200,300,400); the index of the last observation in each.

sorted A matrix giving the order vector for the data. The first column orders by strata, time within strata (longest first), and status within time (censored first). For start, stop data a second column orders by strata, and entry time within strata. The -1 is because subscripts start at 1 in S and 0 in C.

imap matrix containing the indices for random intercepts.

findex a 0/1 matrix with one column for each of fcol and nfrail rows, which marks which coefficients of b are a part of that set. (A bookkeeping array for the C code that is easier to create here.)

P some parameters of the bdsmatrix representing the penalty

The other control parameters are fairly obvious. From this data the C routine can compute the total number of penalized terms and the number that are sparse from the structure of the bdsmatrix, and the total number of intercept terms as max(imap). Other dimensions follow from those.

A dummy call to `kfun` gives the necessary sizes for the penalty matrix. All columns of the stored X matrix are centered and scaled, and these factors are returned. For `theta` we use the first element of each set of initial values found in `itheta`.

⟨*coxfit6a-call*⟩=
```
 if (length(itheta) >0) theta <- sapply(itheta, function(x) x[1])
 else theta <- numeric(0)
 dummy <- kfun(theta, varlist, vparm, ntheta, ncoef)
 if (is.null(dummy@rmat)) rcol <- 0
     else                 rcol <- ncol(dummy@rmat)
 npenal <- ncol(dummy)  #total number of penalized terms

 if (ncol(imap)>0) {
     findex <- matrix(0, nrow=sum(ncoef), ncol=ncol(imap))
```

```
    for (i in 1:ncol(imap)) findex[cbind(imap[,i], i)] <- 1
    }
else findex <- 0  # dummy value

if (is.null(control$sparse.calc)) {
    nevent <- sum(y[,ncol(y)])
    if (length(dummy@blocksize)<=1) nsparse<- 0
    else nsparse <- sum(dummy@blocksize)
    itemp <- max(c(0,imap)) - nsparse  #number of non-sparse intercepts

    if ((2*n) > (nevent*(nsparse-itemp))) control$sparse.calc <- 0
    else control$sparse.calc <- 1
    }

ifit <- .C(Ccoxfit6a,
               as.integer(n),
               as.integer(nvar),
               as.integer(ncol(y)),
               as.double(c(y)),
               as.double(cbind(zmat,x)),
               as.double(offset),
               as.double(weights),
               as.integer(length(newstrat)),
               as.integer(newstrat),
               as.integer(sorted-1),
               as.integer(ncol(imap)),
               as.integer(imap-1),
               as.integer(findex),
               as.integer(length(dummy@blocksize)),
               as.integer(dummy@blocksize),
               as.integer(rcol),
               means = double(nvar),
               scale = double(nvar),
               as.integer(ties=='efron'),
               as.double(control$toler.chol),
               as.double(control$eps),
               as.integer(control$sparse.calc))
means   <- ifit$means
scale   <- ifit$scale
```

The second routine does the real work and is called within the `logfun` function, which is the minimization target of `optim`. The function is called with a trial value of the variance parameters θ, and computes the maximum likelihood estimates of β and b for that (fixed) value of θ, along with the penalized partial likelihood. The normalization constants include the determinant of `kmat`, but since we are using Cholesky decompositions this can be read off of the diagonal. Hopefully the coxvar routines have chosen a parameterization that will mostly avoid invalid

solutions, i.e., those where `kmat` is not symmetric positive definite.

We found that it is best to always do the same number of iterations at each call. Changes in the iteration count (i.e. if one value of θ requires 5 iterations to converge and another only 4 for instance) introduce little 'bumps' into the apparent loglik, which drives `optim` nuts. Hence the min and max iteration count is identical. A similar issue applies to the vector of starting estimates (b, β). It is tempting to use the final results from the prior *theta* evaluation, but again this introduces an artifact. Thus all the calls to `logfun` use the same initial value. Two obvious choices for `init` are a vector of zeros and the fit to a fixed effects model. The latter is likely to be better, but I worry about cases where the fit is nearly singular; user's sometimes fit models with more variables than they should. The current compromise is .7*the final fit + .3*zeros; this number is no more than a wild guess. The addition of (1 -fit0) to the final logliklihood makes the the solution be in the neighborhood of 1 (for the case that the random terms add nothing to the fit) which works well with the convergence criteria of the `optim` routine.

There are actually two C routines `coxfit6b` and `agfit6b`, for ordinary and (start,stop) survival data, respectively. The `timedep` argument is a character string giving the choice.

⟨*define-logfun*⟩=
```
 logfun <- function(theta, varlist, vparm, kfun, ntheta, ncoef,
                    init, fit0, iter, timedep) {
     gkmat <- gchol(kfun(theta, varlist, vparm, ntheta, ncoef))
     if (is.variance) {
         ikmat <- solve(gkmat)  #inverse of kmat, which is the penalty
         if (any(diag(ikmat) <=0)) { #Not an spd matrix
             return(0)  # return a "worse than null" fit
         }
         if (timedep) {
             # start, stop data
             fit <- .Call(Cagfit6b, as.integer(c(iter, iter)),
                          as.double(init), ikmat@blocks, ikmat@rmat)
             }
         else fit <- .Call(Ccoxfit6b, as.integer(c(iter, iter)),
                           as.double(init), ikmat@blocks, ikmat@rmat)
         ilik <- fit$loglik[2] -
             .5*(sum(log(diag(gkmat))) + fit$hdet)
     }
     else {
         # The variance functions have returned the inverse matrix
         if (timedep) fit <- .Call(Cagfit6b, c(iter, iter), as.double(init),
                                   gkmat@blocks, gkmat@rmat)
         else fit <- .Call(Ccoxfit6b, c(iter, iter), as.double(init),
                           gkmat@blocks, gkmat@rmat)
         ilik <- fit$loglik[2] + .5*(sum(log(diag(gkmat))) - fit$hdet)
     }
     -(1+ ilik - fit0)
     }
```

The third routine is used for iterative refinement of the Laplace estimate. The arguments in

this case are

timedep TRUE for start/stop data

beta the final solution vector (b, β), though only β is used.

bmat matrix of trial values for the random coefficients. Should have nfrail rows and refine.n columns.

loglik log-likelihoods at the random points

The routine calculates the log-lik for a succession of Cox models, each one using one of the random draws as it's random effect. The set of trial values is drawn from a t-distribution with refine.df degrees of freedom, centered at the observed random coefficients and with variance hmat-inverse. Now if a random colum vector X has the identity variance matrix, then CX have variance CC'. To get the variance we want we need a matrix C such that $CC' = (H_{bb})^{-1}$ where H_{bb} is the portion of.of the Hessian matrix H corresponding to the random effects. The `bmat` matrix below has such a random sample in each column. We already have the cholesky decompostion LDL' of H in hand; the decomposition of the upper left corner is the upper left corner of the decompostion. The inverse matrix is (L-inverse)' D-inverse (L-inverse), which means we want to backsolve with respect to the upper triangular portion.

The natural way to generate t-variates is to use the mvtnorm library; however, it expects H^{-1} which may be a dense matrix, we already have the sparse cholesky of H `hmat`, and so we essentially duplicate the lines of rmvt and dmvt that occur after matrix decomposition. For further details see the vignette on laplace approximations.

⟨*refine*⟩=
```
 if (refine.n > 0) {
     rdf <- control$refine.df
     nfrail <- ncol(gkmat)
     hmatb <- hmat[1:nfrail, 1:nfrail]
     if (control$refine.method == "control") {
         #create the random t-variate with variance H-inverse
         bmat <- matrix(rnorm(nfrail*refine.n), ncol=refine.n)
         bmat <- backsolve(hmatb, bmat, upper=TRUE) /
             rep(sqrt(rchisq(refine.n, df=rdf)/rdf), each=nfrail)
         bmat2 <- bmat + fit$beta[1:nfrail]  #recenter
     }
     else if (control$refine.method == "direct") {
         bmat <- matrix(rnorm(nfrail*refine.n), ncol=refine.n)
         bmat2 <- gkmat %*% bmat
     }
     else stop("Unrecognized value for refine.method")

     if (timedep) rfit <- .C(Cagfit6d, as.integer(refine.n),
                             as.double(fit$beta),
                             as.double(bmat2),
                             loglik = double(refine.n))
```

```
    else rfit <- .C(Ccoxfit6d, as.integer(refine.n),
                    as.double(fit$beta),
                    as.double(bmat2),
                    loglik = double(refine.n))

    if (control$refine.method == "direct") {
        temp <- max(rfit$loglik)   #keep exp() in range
        errhat <- exp(rfit$loglik - temp)
        mtemp <- mean(errhat)             #estimated integral
        stemp <- sqrt(var(errhat)/refine.n)   #std of the estimate
        r.correct <- c(correction = log(mtemp) + temp -ilik, std= stemp/mtemp)
    }
    else {
        # Penalty terms
        penalty1 <- colSums(bmat2*(ikmat %*% bmat2))/2
        penalty2 <- rowSums((t(bmat) %*% hmatb)^2 )/2

        # Constant for the Gaussian density,  and density of the t-dist (logs)
        gdens <- -0.5* (sum(log(diag(gkmat))) + nfrail*log(2* pi))
        logdet <- -sum(log(diag(hmatb)))
        tdens <- lgamma((nfrail + rdf)/2) -
            (lgamma(rdf/2) + 0.5*(logdet + nfrail* log(pi*rdf) +
                                  (nfrail+ rdf)* log(1 + 2*penalty2/rdf)))

        # Add it up, we have to be very careful about round-off
        n1 <- rfit$loglik + gdens - (penalty1 + ilik + tdens)
        n2 <- fit$loglik[2] + gdens - (penalty2 + ilik + tdens)
        temp <- max(n1, n2)  #scale so the largest value is about 1
        errhat <- (exp(n1-temp) - exp(n2-temp)) * exp(temp)
        #errhat <- (exp(rfit$loglik -(penalty1 + ilik)) -
        #        exp(fit$loglik[2]- (penalty2 + ilik))) * exp(gdens-tdens)

        mtemp <- mean(errhat)             #estimated integral
        stemp <- sqrt(var(errhat)/refine.n)   #std of the estimate
        r.correct <- c(correction= log(1+ mtemp), std=stemp/(1 +mtemp))
    }
}
```

The final routine `coxfit6c` is used for cleanup, and is described in section 5.5.

5.3 Setup

Preliminaries aside, let's now build the routine. The input arguments are as were set up by `coxme`, this routine would never be called directly by a user.

x the matrix of fixed effects

y the survival times, an object of class 'Surv'

strata strata vector

offset vector of offsets, usually all zero

control the result of a call to coxme.control

weights vector of case weights. usually 1

ties the method for handling ties, 'breslow' or 'efron'

rownames needed for labeling the output, in the rare case that the X matrix is null.

imap matrix of random factor (intercepts) indices. If imap[4,1]=6, imap[4,2]=10 this means that observation 4 contributes to both coefficient 6 and coefficient 10, both of which are random intercepts.

zmat the Z matrix, the design matrix for random slopes

varlist the list describing the structure of the random effects

vparm the list of parameters for the variance functions

itheta initial values for the random effects, e.g., the ones we need to solve for (may be null if the variances are all fixed)

ntheta vector giving the number of thetas for each random term

refine.n number of iterations for iterative refinement

⟨*coxme-setup*⟩=
```
 coxme.fit <- function(x, y, strata, offset, ifixed, control,
                         weights, ties, rownames,
                         imap, zmat, varlist, vparm, itheta,
                         ntheta, ncoef, refine.n, is.variance) {
     #     time0 <- proc.time() #debugging line
     n <-  nrow(y)
     if (length(x) ==0) nvar <-0
     else nvar <- ncol(as.matrix(x))

     if (missing(offset) || is.null(offset)) offset <- rep(0.0, n)
     if (missing(weights)|| is.null(weights))weights<- rep(1.0, n)
     else {
         if (any(weights<=0)) stop("Invalid weights, must be >0")
         }
```

The next step is to get a set of sort indices, but not to actually sort the data. This was a key insight which allows the (start,stop) version to do necessary bookkeeping in time of $(2n)$ instead of $O(n^2)$. We sort by strata, time within strata (longest first), and status within time (censor before deaths). For (start, stop) data a second index orders the entry times.

⟨*coxme-setup*⟩=
```
    if (ncol(y) ==3) {
        if (length(strata) ==0) {
            sorted <- cbind(order(-y[,2], y[,3]),
                            order(-y[,1]))
            newstrat <- n
            }
        else {
            sorted <- cbind(order(strata, -y[,2], y[,3]),
                            order(strata, -y[,1]))
            newstrat  <- cumsum(table(strata))
            }
        status <- y[,3]
        timedep <- TRUE
        coxfitfun<- agreg.fit
        }
    else {
        if (length(strata) ==0) {
            sorted <- order(-y[,1], y[,2])
            newstrat <- n
            }
        else {
            sorted <- order(strata, -y[,1], y[,2])
            newstrat <-  cumsum(table(strata))
            }
        status <- y[,2]
        timedep <- FALSE
        coxfitfun <- coxph.fit
        }
```

The last step of the setup is to do an initial fit. We want two numbers: the loglik for a no-random-effects and initial-values-for-fixed (usually 0) fit, and that for the best fixed effects fit. The first is the NULL model loglik for the fit as a whole, the second is used to scale the logliklihood during iteration, the `fit0` parameter in the `logfun` function. The easiest way to get these is from an ordinary `coxph` call. Most coxph calls converge in 3-4 iterations. The default value for `control$inner.iter` is `Quote(fit0$iter +1)` to avoid disaster in the case of a 'hard' baseline model. We need to evaluate the expression after fit0 is known. If all values of θ are fixed, then the only thing we will use from fit0 is the loglik. Note that if there are no covariates or only an offset term, then the returned log-likelihood is of length 1, not 2. a null model.

⟨*null-fit*⟩=
```
 if (is.null(ifixed) ) {
     ifixed <- rep(0., ncol(x))
     if (length(ifixed) ==0) ifixed <- NULL  #agreg.fit didn't like numeric(0)
 }
 else if (length(ifixed) != ncol(x))
     stop("Wrong length for initial parameters of the fixed effects")
```

```
if (length(itheta)==0) itemp <- 0 else itemp <- control$iter.max
fit0 <- coxfitfun(x,y, strata=strata,
                  offset=offset, init=ifixed, weights=weights,
                  method=ties, rownames=1:nrow(y),
                  control=coxph.control(iter.max=itemp))
loglik0 <- fit0$loglik[length(fit0$loglik)]  # in case of no covariates
control$inner.iter <- eval(control$inner.iter)
```

5.4 Doing the fit

If there are any parameters to optimize over, we now do so. Our last step before optimization is to set the starting value. We will have inherited a list of possible starting values for each parameter in `istart`; try all combinations and keep the best one.

⟨*coxme-fit*⟩=
```
⟨define-logfun⟩

ishrink <- 0.7  # arbitrary guess
init.coef <- c(rep(0., npenal), scale*fit0$coef* ishrink)

if (length(itheta)==0) iter <- c(0,0)
else {
    ⟨coxme-gridsearch⟩
```

This is set out as a block since it is also used in lmekin. (Later, copied but not used due to different logfun).

⟨*coxme-gridsearch*⟩=
```
nstart <- sapply(itheta, length)
if (all(nstart==1)) theta <- unlist(itheta)  #one starting guess
else {
    #make a matrix of all possible starting estimtes
    testvals <- do.call(expand.grid, itheta)
    bestlog <- NULL
    for (i in 1:nrow(testvals)) {
        ll <- logfun(as.numeric(testvals[i,]),
                     varlist, vparm, kfun, ntheta, ncoef,
                     init=init.coef, loglik0,
                     control$inner.iter, timedep)
        if (is.finite(ll)) {
            #ll calc can fail if someone picks a very bad starting guess
            if (is.null(bestlog) || ll < bestlog) {
                # (optim is set up to minimize)
                bestlog <- ll
                theta <- as.numeric(testvals[i,])
            }
```

```
        }
    }
    if (is.null(bestlog))
        stop("No starting estimate was successful")
}
```

In the code below `optpar` is a list of control parameters for the `optim` function, which are defined in `coxme.control` and accessible for the user to change, and `logpar` is a list of parameters that will be needed by logfun. In R the ones that are simple copies such as `timedep` would not need to be included in the list since they are inherited with the environment, however, I prefer to make such hidden arguments explicit.

⟨*coxme-fit*⟩=
```
    # Finally do the fit
    logpar <- list(varlist=varlist, vparm=vparm,
                   ntheta=ntheta, ncoef=ncoef, kfun=kfun,
                   init=init.coef, fit0= loglik0,
                   iter=control$inner.iter,
                   timedep = timedep)
    mfit <- do.call('optim', c(list(par= theta, fn=logfun, gr=NULL),
                           control$optpar, logpar))
    theta <- mfit$par
    iter <- mfit$counts[1] * c(1, control$inner.iter)
}
```

The optimization finds the best value of theta, but does not return all the parameters we need from the fit. So we make one more call. This is essentially the "inside" of `logfun`. The phrase `c(ikmat@rmat,0)` makes sure something is passed when `rmat` is of length 0.

⟨*coxme-fit*⟩=
```
gkmat <- gchol(kfun(theta, varlist, vparm, ntheta, ncoef))
if (is.variance) {
    ikmat <- solve(gkmat)  #inverse of kmat, which is the penalty
    if (timedep)
        fit <- .Call(Cagfit6b, iter= as.integer(c(0L, control$iter.max)),
                     beta <- c(rep(0., npenal), fit0$coef*scale),
                     ikmat@blocks, c(ikmat@rmat, 0.))
    else fit <- .Call(Ccoxfit6b, iter= as.integer(c(0L, control$iter.max)),
                      beta <- c(rep(0., npenal), fit0$coef*scale),
                      ikmat@blocks, c(ikmat@rmat, 0.))
    ilik <- fit$loglik[2] -
        .5*(sum(log(diag(gkmat))) + fit$hdet)
} else {
    if (timedep)
        fit <- .Call(Cagfit6b, iter= as.integer(c(0L, control$iter.max)),
                     beta <- c(rep(0., npenal), fit0$coef*scale),
                     gkmat@blocks, c(gkmat@rmat, 0.))
```

```
    else fit <- .Call(Ccoxfit6b, iter= as.integer(c(0L, control$iter.max)),
                      beta <- c(rep(0., npenal), fit0$coef*scale),
                      gkmat@blocks, c(gkmat@rmat, 0.))
    ilik <- fit$loglik[2] +
        .5*(sum(log(diag(gkmat))) - fit$hdet)
}
iter[2] <- iter[2] + fit$iter
```

5.5 Finishing up

There are 6 tasks left to do

```
⟨coxme-finish⟩=
 ⟨coxme-lastvar⟩
 ⟨coxme-rescale⟩
 ⟨coxme-df⟩
 ⟨refine⟩
 .C(Ccoxfit6e, as.integer(ncol(y)))  #release memory
 ⟨create-output-list⟩
```

The nexts section finishes up with the C code. The first few lines reprise some variables found in the C code but not before needed here. It returns the score vector u, the sparse and dense portions of the Cholesky decomposition of the Hessian matrix (h.b and h.r), the inverse Hessian matrix (hi.b, hi.r), and the rank of the final solution. These are needed to compute the variance matrix of the estimates.

```
⟨coxme-lastvar⟩=
 nfrail <- nrow(ikmat)  #total number of penalized terms
 nsparse <- sum(ikmat@blocksize)
 nvar2  <- nvar + (nfrail - nsparse)  # total number of non-sparse coefs
 nvar3  <- as.integer(nvar + nfrail)  # total number of coefficients
 btot   <- length(ikmat@blocks)

 fit3 <- .C(Ccoxfit6c,
                u    = double(nvar3),
                h.b  = double(btot),
                h.r  = double(nvar2*nvar3),
                hi.b = double(btot),
                hi.r = double(nvar2*nvar3),
                hrank= integer(1),
                as.integer(ncol(y))
                )
```

Now create the Hessian and inverse Hessian matrices; the latter of these is the variance matrix. The C code had centered and rescaled all X matrix coefficients so we need to undo that scaling. First we deal with a special case, if there are only sparse terms then `hmat` and `hinv` have only a block-diagonal component. (This happens more often than you might think, a random per-subject intercept for instance.)

⟨*coxme-rescale*⟩=

```
    if (nvar2 ==0) {
        hmat <- new('gchol.bdsmatrix', Dim=c(nvar3, nvar3),
                    blocksize=ikmat@blocksize, blocks=fit3$h.b,
                    rmat=matrix(0,0,0), rank=fit3$hrank,
                    Dimnames=list(NULL, NULL))
        hinv <- bdsmatrix(blocksize=ikmat@blocksize, blocks=fit3$hi.b)
        }
```

And now three cases: no X variables, a single X, or multiple X variables. Assume there are p=nvar variables and let `V` be the lower $p \times p$ portion of the `nvar3` by `nvar2` R matrix, and S = `diag(scale` be the rescaling vector. X was replaced by XS^{-1} before computation. For the Hessian, we want to replace V with SVS and for the inverse hessian with $S^{-1}VS^{-1}$. The matrix `hmat` is however a Cholesky decomposition of the hessian $H = LDL'$ where L is lower triangular with ones on the diagonal and D is diagonal; D is kept on the diagonal of V and L below the diagonal. A little algebra shows that we want to replace D (the diagonal of L) with S^2D and L with SLS^{-1}.

⟨*coxme-rescale*⟩=

```
    else {
        rmat1 <- matrix(fit3$h.r, nrow=nvar3)
        rmat2 <- matrix(fit3$hi.r, nrow=nvar3)
        if (nvar ==1 ) {
            rmat1[nvar3,] <- rmat1[nvar3,]/scale
            rmat2[nvar3,] <- rmat2[nvar3,]/scale
            rmat1[,nvar2] <- rmat1[,nvar2]*scale
            rmat2[,nvar2] <- rmat2[,nvar2]/scale
            rmat1[nvar3,nvar2] <- rmat1[nvar3,nvar2]*scale^2
            u <- fit3$u  # the efficient score vector U
            u[nvar3] <- u[nvar3]*scale
            }
        else if (nvar >1) {
            temp <- seq(to=nvar3, length=length(scale))
            u <- fit3$u
            u[temp] <- u[temp]*scale
            rmat1[temp,] <- (1/scale)*rmat1[temp,] #multiply rows* scale
            rmat2[temp,] <- (1/scale)*rmat2[temp,]

            temp <- temp-nsparse          #multiply cols
            rmat1[,temp] <- rmat1[,temp] %*% diag(scale)
            rmat2[,temp] <- rmat2[,temp] %*% diag(1/scale)
            temp <- seq(length=length(scale), to=length(rmat1), by=1+nvar3)
            rmat1[temp] <- rmat1[temp]*(scale^2)    #fix the diagonal
            }
        hmat <- new('gchol.bdsmatrix', Dim=c(nvar3, nvar3),
                    blocksize=ikmat@blocksize, blocks=fit3$h.b,
                    rmat= rmat1, rank=fit3$hrank,
```

```
                    Dimnames=list(NULL, NULL))
        hinv <- bdsmatrix(blocksize=ikmat@blocksize, blocks=fit3$hi.b,
                          rmat=rmat2)
        }
```

Now for the degrees of freedom, using formula 5.16 of Therneau and Grambsch. First we have a small utility function to compute the trace(AB) where A and B are bdsmatrix objects. For ordinary matrices this is the sum of the element-wise product of A and B', but we have to account for the fact that bdsmatrix objects only keep the lower diagonal of the block portion. We need the diagonal sum + 2 times the off-diagonal sum.

⟨*coxme-df*⟩=
```
 traceprod <- function(H, P) {
     #block-diagonal portions will  match in shape
     nfrail <- nrow(P)  #penalty matrix
     nsparse <- sum(P@blocksize)
     if (nsparse >0) {
         temp1 <- 2*sum(H@blocks * P@blocks) -
                  sum(diag(H)[1:nsparse] * diag(P)[1:nsparse])
         }
     else temp1 <- 0

     if (length(P@rmat) >0) {
         #I only want the penalized part of H
         rd <- dim(P@rmat)
         temp1 <- temp1 + sum(H@rmat[1:rd[1], 1:rd[2]] * P@rmat)
         }
     temp1
     }

 df <- nvar + (nfrail - traceprod(hinv, ikmat))
```

And last, put together the output structure.

⟨*create-output-list*⟩=
```
 idf <- nvar + sum(ntheta)
 fcoef <- fit$beta[1:nfrail]
 penalty <- sum(fcoef * (ikmat %*% fcoef))/2

 if (nvar > 0) {
     out <- list(coefficients = fit$beta[-(1:nfrail)]/scale, frail=fcoef,
          theta=theta, penalty=penalty,
          loglik=c(fit0$log[1], ilik, fit$log[2]), variance=hinv,
          df=c(idf, df), hmat=hmat, iter=iter, control=control,
          u=u, means=means, scale=scale)
     }
 else out <- list(coefficients=NULL, frail=fcoef,
```

```
                theta=theta, penalty=penalty,
          loglik=c(fit0$log[1], ilik, fit$log[2]), variance=hinv,
          df=c(idf, df), hmat=hmat, iter=iter, control=control,
          u=fit3$u, means=means, scale=scale)

if (refine.n>0) {
    out$refine <- r.correct
    #The next line can be turned on for detailed tests in refine.R
    #  The feature is not documented in the manual pages, only
    #  here.
    if (control$refine.detail) {
        if (control$refine.method== "control")
            out$refine.detail <-list(loglik=rfit$loglik, bmat=bmat2,
                                     tdens=tdens,
                                     penalty1=penalty1, penalty2=penalty2,
                                     gdens=gdens, errhat=errhat, gkmat=gkmat)
        else out$refine.detail <- list(loglik=rfit$loglik, bmat=bmat2,
                                       errhat=errhat, gkmat=gkmat)
    }
}
out
```

6 Methods for the random effects

Creating methods for the random effects turned out to be tricky. The problem is that I want to play along with nlme and lme4.

The nlme package defines `ranef`, `random.effects`, `fixef`, and `fixed.effects` as standard S3 generics. If nlme is loaded first, I don't want to re-define these functions. If I do, then the ranef.lme method becomes invisible. It appears to be a design decision: R doesn't know that my ranef function is identical to the ones in nlme, and so it 'forgets' the old methods in order to avoid inconsistency. The obvious solution to this is to check for existence of the functions before defining them. However, this doesn't work with namespaces – you either have the file listed for export in the NAMESPACE file or you don't.

If nlme is loaded after coxme, there will be a set of messages about replacement of the 4 functions; there is nothing I can do about that. However, my definitions are now forgotton. A solution to this is to make ranef.coxme and fixef.coxme exported symbols in the name space. R now finds them by standard mechanisms outside the name space structure.

After some discussion on the R developer list, it was decided that the only workable solution was to include the line

importFrom(nlme, ranef, fixef, VarCorr)

into both coxme and lme4, importing the generic from the original nlme package. It is the only way for the R exectutive to know that all the instances of a method are legal.

⟨*ranef*⟩=
```
 # The objects that do the actual work (not much work)
```

```
fixef.coxme <- function(object, ...)
    object$coefficients

fixef.lmekin <- function(object, ...)
    object$coefficients$fixed

ranef.coxme <- function(object, ...)
    object$frail

ranef.lmekin <- function(object, ...)
    object$coefficients$random

VarCorr.coxme <- function(x, ...)
    x$vcoef

VarCorr.lmekin <- function(x, ...)
    x$vcoef

vcov.coxme <- function(object, ...) {
    nf <- length(fixef(object))
    indx <- seq(length=nf, to=nrow(object$var))
    as.matrix(object$var[indx, indx])
}

vcov.lmekin <- vcov.coxme
```

For the logLik method we give the number of events as the number of observations.

⟨*ranef*⟩=

```
logLik.coxme <- function(object, type=c("penalized", "integrated"), ...) {
    type <- match.arg(type)
    if (type=='penalized') {
        out <- object$loglik[3] + object$penalty
        attr(out, "df") <- object$df[2]
        }
    else {
        out <- object$loglik[2]
        attr(out, "df") <- object$df[1]
        }
    attr(out, "nobs") <- object$n[1]  #number of events
    class(out) <- "logLik"
    out
}

logLik.lmekin <- function(object, ...) {
    out <- object$loglik[2]
    attr(out, "df") <- object$df[1]
```

```
    class(out) <- "logLik"
    out
    }
```

7 lmekin

The original kinship library had an implementation of linear mixed effects models using the matrix code found in coxme. The reason for the program was entirely to check our arithmetic: it should get the same answers as lme. With more time and a larger test suite the routine is no longer necessary for this purpose, and I intended to retire it. However, it had became popular with users since it can fit a few models that lme cannot, so now is a permanent part of the package.

The orignal code was based on equation 2.14 of Pinheiro and Bates, the one they do not recommmend for computation. This is a quite sensible formula if there is a *single* random effect but it does not generalize well. This release follows the compuational strategy of lme much more closely. Note that a lot of the code below is a pure copy of the coxme code.

```
⟨lmekin⟩=
 lmekin <- function(formula,  data,
         weights, subset, na.action,
         control, varlist, vfixed, vinit,
         method=c("ML", "REML"),
         x=FALSE, y=FALSE, model=FALSE,
         random, fixed, variance,  ...) {

    Call <- match.call()
    sparse <- c(1,0)  #needed for compatablily with coxme code
    ⟨lme-process-standard-arguments⟩
    ⟨decompose-lme-formula⟩
    ⟨build-control-structures⟩
    ⟨lmekin-compute⟩
    ⟨lmekin-finish-up⟩
    }
 ⟨lmekin-helper⟩
```

The standard argments processing is copy of that for coxme, but with the word "lmekin" in error messages.

```
⟨lme-process-standard-arguments⟩=
 if (!missing(fixed)) {
    if (missing(formula)) {
        formula <- fixed
        warning("The 'fixed' argument of lmekin is depreciated")
        }
    else stop("Both a fixed and a formula argument are present")
    }
```

```
if (!missing(random)) {
    warning("The random argument of lmekin is depreciated")
    if (class(random) != 'formula' || length(random) !=2)
        stop("Invalid random formula")
    j <- length(formula)   #will be 2 or 3, depending on if there is a y

    # Add parens to the random formula and paste it on
    formula[[j]] <- call('+', formula[[j]], call('(', random[[2]]))
    }

if (!missing(variance)) {
    warning("The variance argument of lmekin is depreciated")
    vfixed <- variance
    }

method <- match.arg(method)

temp <- call('model.frame', formula= subbar(formula))
for (i in c('data', 'subset', 'weights', 'na.action'))
    if (!is.null(Call[[i]])) temp[[i]] <- Call[[i]]
m <- eval.parent(temp)

Y <- model.extract(m, "response")
n <- length(Y)
if (n==0) stop("data has no observations")

weights <- model.weights(m)
if (length(weights) ==0) weights <- rep(1.0, n)
else if (any(weights <=0))
    stop("Negative or zero weights are not allowed")

offset <- model.offset(m)
if (length(offset)==0) offset <- rep(0., n)

# Check for penalized terms; the most likely is pspline
pterms <- sapply(m, inherits, 'coxph.penalty')
if (any(pterms)) {
    stop("You cannot have penalized terms in lmekin")
    }

if (missing(control)) control <- lmekin.control(...)
```

Get the X-matrix part of the formula. This is parallel to the version in coxme, the main difference is that we keep the intercept term. We check for the cluster and strata terms because it is a mistake that I anticipate users to make.

⟨*decompose-lme-formula*⟩=

```
flist <- formula1(formula)
if (hasAbar(flist$fixed))
    stop("Invalid formula: a '|' outside of a valid random effects term")

special <- c("strata", "cluster")
Terms <- terms(flist$fixed, special)
if (length(attr(Terms, "specials")$strata))
    stop ("A strata term is invalid in lmekin")
if (length(attr(Terms, "specials")$cluster))
    stop ("A cluster term is invalid in lmekin")
X <- model.matrix(Terms, m)
```

Now for the actual compuation. We want to solve

$$\begin{aligned} y &= X\beta + Zb + \epsilon \\ b &\sim N(0, \sigma^2 K) \\ \epsilon &\sim N(0, \sigma^2) \end{aligned}$$

where K is the variance matrix returned by `kfun`. If we know K, one way to solve this is as an augmented least squares problem with

$$y^* = \begin{pmatrix} y \\ 0 \end{pmatrix} \qquad X^* = \begin{pmatrix} X \\ 0 \end{pmatrix} \qquad Z^* = \begin{pmatrix} Z \\ \Delta \end{pmatrix}$$

where $\Delta'\Delta = K^{-1}$. The dummy rows of data have $y = 0$, $X = 0$ and Δ as the predictor variables. With known Δ, this gives the solution to all the other parameters as an ordinary least squares problem. If $K = U'U$ for U an upper triangular matrix, then $K^{-1} = L'L$ where $L = (U')^{-1} = (U^{-1})'$ is lower triangular. Then

$$\begin{aligned} \Delta'\Delta &= K^{-1} \\ \Delta &= L \end{aligned} \tag{2}$$

In our case K will be the iteration target of the `optim` function, and we need to evaluate the other parameters in order to determine the log-likelihood. In coxme this is done inside a C routine, here we can use the more direct method. In the original `lmekin` function we made the assumption that Z was an identity matrix, which allowed for a simple solution using only the generalized cholesky decompostion found in the `bdsmatrix` library. Here we use the more general QR method as outlined in Pinheiro and Bates. Assume that Z has q columns and X has p columns, the number of random and fixed coefficients, respectively. Then

$$\begin{aligned} (Z^*, X^*) &= QR \\ R &= \begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \\ 0 & 0 \end{pmatrix} \\ Q'y &= \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} \end{aligned}$$

The orthagonal matrix Q is $n \times n$, R_{11} is $q \times q$ and upper triangular, R_{22} is $p \times p$ upper triangular, and R is $n \times p$. The vectors c_1, c_2, and c_3 are of lengths q, p, and $n - (p + q)$, respectively. Using slightly different notation, Pinheiro and Bates show that the solution vector and the profiled log-likelihood are (equations 2.19 and 2.21)

$$\hat{\beta}(\theta) = R_{22}^{-1} c_2 \tag{3}$$

$$\hat{\sigma}^2(\theta) = ||c_3||^2/n \tag{4}$$

$$\log(L(\theta)) = \frac{n}{2}\left[\log n - \log(2\pi) - 1\right] - n \log |c_2| + \log\left(\text{abs}\frac{|\Delta|}{|R_{11}}\right) \tag{5}$$

Here $|c|$ is the norm of a vector c and $|A|$ the determinant of a matrix A. The determinant of a triangular matrix is the product of its diagonal elements. The solution for $\hat{\beta}$ is returned by the `qr.coef` routine.

The restricted maximum likelihood estimate (REML) follows from the same decompostions, but with

$$\hat{\sigma}^2_{REML}(\theta) = ||c_3||^2/(n-p) \tag{6}$$

$$\log(L(\theta))_{REML} = \frac{n}{2}\left[\log n - \log(2\pi) - 1\right] - (n-p) \log |c_2| + \log\left(\text{abs}\frac{|\Delta|}{|R|}\right) \tag{7}$$

⟨ *lmekin-compute* ⟩=
 ⟨ *define-penalty* ⟩
 ⟨ *define-xz* ⟩
 ⟨ *lmekin-fit* ⟩

The define-penalty code is shared with coxme, it defines the function `kfun` which returns K/σ^2 given the parameters θ. The next bit of code defines X^* and the top portion of Z^* as sparse Matrix objects. The definition X^* is easy as we already have it in hand. For Z^* most of the work is creating the design matrix for the intercepts from our very compressed form `fmat`. That matrix has one column for each unique factor and n rows, each column contains the coefficent mapping of subjects to coefficients. So for instance assume 6 subject and a term of `(1|group)` with 3 groups. The corresponding column of fmat might be (1,2,2,3,1,3) showing that subject 1 is in group 1, subject 1 in group 2, etc. In a genetic data set with kinship each subject would be in thier own group and the column would be some permutation of 1:n. The corresponding design matrix is

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The sparse coding of this for a dgCMatrix object has components

i a vector containing all the row numbers of the non zero elements, with rows numbered from zero. In this case it would be 0, 4, 1, 2, 3, 5.

p a vector with first element 0 such that diff(p) = the number of non-zeros in each column

x the values of the non-sparse elements

Dim dimensions of the matrix

Dimnames optional dimnames

factors an empty list, used by later Matrix routines for factorization information

The variable `zstar1` is the top part of Z^*, i.e., the full Z matrix, in sparse form. At each iteration Δ changes, we splice that on at that time. If we use a decompostion of (Z^*, X^*) as defined above there is a problem: the sparse QR routine will rearrange the columns of the decomposed matrix so as to be most efficient (some permutataions retain more sparseness than others). This is ok as long as the rearrangement does not intermix Z and X, and in fact it often does not since Z will be "sparser" than X in most problems; but we can't guarrantee it. Thus we do a two-step decompostion:

$$(Z, X) = (Q_1|Q_2) \begin{pmatrix} R_1 & A \\ 0 & R2 \end{pmatrix}$$
$$Q_1'X = \begin{pmatrix} A \\ Q_2 R_2 \end{pmatrix}$$

Thus Q_1 and R_1 are the result of a QR decomposition of Z, and Q_2, R_2 from a QR decompostion of the the lower rows of $Q_1'X$. The final result is the same as a single QR call for the combined matrix.

At the time of this writing the Matrix library's `qr.qty` routine could not deal with a sparse matrix as the second argument, thus in creating `xstar` below we force a non-sparse version. This is not a computational problem since X is always of modest size, it is the random effects matrix Z which can be huge and for which sparseness can pay off handsomely. The Matrix routine by default uses a sparse form if the object has over 1/2 zeros, which would be true for some X matrices.

⟨*define-xz*⟩=
```
 #Define Z^* and X^*
 itemp <- split(row(fmat), fmat)
 zstar1 <- new("dgCMatrix",
               i= as.integer(unlist(itemp) -1),
               p= as.integer(c(0, cumsum(unlist(lapply(itemp, length))))),
               Dim=as.integer(c(n, max(fmat))),
               Dimnames= list(NULL, NULL),
               x= rep(1.0, length(fmat)),
               factors=list())
 if (length(zmat) >0)  {
     # there were random slopes as well
     zstar1 <- cbind(zstar1, as(Matrix(zmat), "dgCMatrix"))
 }
```

```
nfrail <- ncol(zstar1)
nvar <- ncol(X)
if (nvar == 0)  xstar <- NULL  #model with no covariates
else xstar <- rbind(Matrix(X, sparse=FALSE),
                    matrix(0., nrow=nfrail, ncol=ncol(X)))
ystar <- c(Y, rep(0.0, nfrail))
```

Now to do the fit. Define logfun, which returns the loglik (without the constant terms) for a given trial value of theta. Use a gridsearch to find the best starting values, and start the optim() routine there. The convergence criterion for optim works well if the true minimum is around 1 in absolute value; our last line of logfun makes that true if the starting estimate is exactly the final solution. Notice that the max for θ only depends on the loglik, equation 5 or 7.

For the ML estimate 5 we need the determinant of R_{11} The documentation for the qr routine in the Matrix library has an unclear reference to column permutations (it says they can exist, but not how to turn this on or off nor the default). Since we need to keep Z before X the code below has 2 calls, first on the Z portion and then on the transformed X portion.

A second nuisance is that the qr.R function in the Matrix library insists on printing a warning message about the fact that permutations may exist. For computation of a determinant, which is the product of the diagonal elements of R, any reordering is irrelevant to us. We use a local function mydiag to work around this.

To create *Delta* first do the cholesky decompostion, which returns the upper triangular matrix U by default. The solve function when applied to the cholesky uses a fast backsolve approach. (But you can't use the backsolve function, since the Matrix library didn't have a method for it at this time. I originally tried this to make the code clearer wrt to the algorithm.) Per equaton (2) we need to transpose the result to lower triangular form.

⟨*lmekin-fit*⟩=

```
mydiag <- function(x) {
    if (class(x)=="sparseQR") diag(x@R)
    else diag(qr.R(x))
}

logfun <- function(theta, best=0) {
    vmat <- kfun(theta, varlist, vparm, ntheta, ncoef)
    Delta <- t(solve(chol(as(vmat, "dsCMatrix"), pivot=FALSE)))
    zstar <- rbind(zstar1, Delta)
    qr1 <- qr(zstar)
    dd <- mydiag(qr1)
    cvec <- as.vector(qr.qty(qr1, ystar))[-(1:nfrail)]  #residual part
    if (nvar >0) {  # have covariates
        qr2 <- qr(qr.qty(qr1, xstar)[-(1:nfrail),])
        cvec <- qr.qty(qr2, cvec)[-(1:nvar)]  #residual part
        if (method!= "ML") dd <- c(dd, mydiag(qr2))
    }

    loglik <- sum(log(abs(diag(Delta)))) - sum(log(abs(dd)))
```

```
    if (method=="ML") loglik <- loglik - .5*n*log(sum(cvec^2))
    else              loglik <- loglik - .5*length(cvec)*log(sum(cvec^2))

    best - (loglik+1)  #optim() wants to minimize rather than maximize
}

nstart <- sapply(itheta, length)
if (length(nstart) ==0) theta <- NULL #no thetas to solve for
else {
    #iteration is required
    #make a matrix of all possible starting estimtes
    testvals <- do.call(expand.grid, itheta)
    bestlog <- NULL
    for (i in 1:nrow(testvals)) {
        ll <- logfun(as.numeric(testvals[i,]))
        if (is.finite(ll)) {
            #ll calc can fail if someone picks a very bad starting guess
            if (is.null(bestlog) || ll < bestlog) {
                # (optim is set up to minimize)
                bestlog <- ll
                theta <- as.numeric(testvals[i,])
            }
        }
    }
    if (is.null(bestlog))
        stop("No starting estimate was successful")

    optpar <- control$optpar
    optpar$hessian <- TRUE
    mfit <- do.call('optim', c(list(par= theta, fn=logfun, gr=NULL,
                                    best=bestlog), optpar))
    theta <- mfit$par
}
```

At this point the optimal θ has been found. Now do one more pass with the "internals" of the logfun function, and compute other quantities that we didn't need for the intermediate iterations. Remember that lmekin was designed for genetic problems, and for these Z will be very large and sparse while X will be modest.

⟨*lmekin-compute*⟩=
```
vmat <-  kfun(theta, varlist, vparm, ntheta, ncoef)
Delta <- t(solve(chol(as(vmat, "dsCMatrix"), pivot=FALSE)))
zstar <- rbind(zstar1, Delta)
qr1 <- qr(zstar)
dd <- mydiag(qr1)
ctemp <- as.vector(qr.qty(qr1, ystar))
cvec <- ctemp[-(1:nfrail)]  #residual part
```

```
if (is.null(xstar)) { #No X covariates
    rcoef <- qr.coef(qr1, ystar)
    yhat <- qr.fitted(qr1, ystar)
}
else {
    qtx <- qr.qty(qr1, xstar)
    qr2 <- qr(qtx[-(1:nfrail),,drop=F])
    if (method!="ML") dd <- c(dd, mydiag(qr2))

    fcoef <-qr.coef(qr2, cvec)
    yresid <- ystar - xstar %*% fcoef
    rcoef <- qr.coef(qr1, yresid)
    cvec <- qr.qty(qr2, cvec)[-(1:nvar)] #residual part
    if (class(qr2)=="sparseQR") varmat <- chol2inv(qr2@R)
    else varmat <- chol2inv(qr.R(qr2))
    yhat <- as.vector(zstar1 %*% rcoef + X %*% fcoef) #kill any names
}

if (method=="ML") {
    sigma2 <- sum(cvec^2)/n  #MLE estimate
    loglik <- sum(log(abs(diag(Delta)))) -
          (sum(log(abs(dd))) + .5*n*(log(2*pi) +1 + log(sigma2)))
}
else {
    np <- length(cvec)  # n-p
    sigma2 <- mean(cvec^2)  # divide by n-p
    loglik <- sum(log(abs(diag(Delta)))) -
        (sum(log(abs(dd))) + .5*np*(log(2*pi) + 1+ log(sigma2)))
}

# Debugging code, set the argument to TRUE only during testing
if (FALSE) {
    # Compute the alternate way (assumes limited reordering)
    zx <- cbind(zstar, as(xstar, class(zstar)))
    qr3 <- qr(zx)
    cvec3 <- qr.qty(qr3, ystar)[-(1:(nvar+nfrail))]
    if (method=="ML")  dd3 <- (diag(myqrr(qr3)))[1:nfrail]
    else               dd3 <- (diag(myqrr(qr3)))[1:(nfrail+nvar)]
    #all.equal(dd, dd3)
    #all.equal(cvec, cvec3)
    acoef <- qr.coef(qr3, ystar)
    browser()
}
```

Bundle the results together into an output object. This object differs from the old lme object,

having both more and less information. First we call the wrapup functions to retransform any parameters. At this point we also rescale the other variance components: the iteration used $\sigma^2\Sigma$ as the variance matrix for the random effects, the user wants to think of Σ as the product of these.

⟨ *lmekin-finish-up* ⟩=

```
newtheta <- random.coef <- list()
nrandom <- length(varlist)
sindex <- rep(1:nrandom, ntheta) #which thetas to which terms
bindex <- rep(1:nrandom, rowSums(ncoef)) # which b's to which terms
for (i in 1:nrandom) {
    temp <- varlist[[i]]$wrapup(theta[sindex==i], rcoef[bindex==i],
                                vparm[[i]])
    newtheta <- c(newtheta, lapply(temp$theta, function(x) x*sigma2))
    if (!is.list(temp$b)) {
        temp$b <- list(temp$b)
        names(temp$b) <- paste("Random", i, sep='')
        }
    random.coef <- c(random.coef, temp$b)
    }
```

We create a variance matrix only for the fixed effects. The primary reason is that even though Z is sparse the variance matrix associated with Z will usually be dense; for many of our genetics problems this would easily drive R out of memory. If the columns of (Z, X) remain in order we only need to invert the lower triangle of R, but if they have been permuted we need to force separation between Z and X by doing the decompostion in two steps.

⟨ *lmekin-finish-up* ⟩=

```
if (length(fcoef) >0) {
    # There are fixed effects
    nvar <- length(fcoef)
    fit <- list (coefficients=list(fixed=fcoef, random=random.coef),
                 var = varmat * sigma2,
                 vcoef =newtheta,
                 residuals= Y- yhat,
                 method=method,
                 loglik=loglik,
                 sigma=sqrt(sigma2),
                 n=n,
                 call=Call)
}
else fit <- list(coefficients=list(fixed=NULL, random=random.coef),
                 vcoef=newtheta,
                 residuals=Y - yhat,
                 method=method,
```

```
                  loglik=loglik,
                  sigma=sqrt(sigma2),
                  n=n,
                  call=Call)

 if (!is.null(theta)) {
     fit$rvar <- mfit$hessian
     fit$iter <- mfit$counts
 }
 if (x) fit$x <- X
 if (y) fit$y <- Y
 if (model) fit$model <- m

 na.action <- attr(m, "na.action")
 if (length(na.action)) fit$na.action <- na.action

 class(fit) <- "lmekin"
 fit
```

And last, a couple of helper functions

⟨*lmekin-helper*⟩=
```
 residuals.lmekin <- function(object, ...) {
     if (length(object$na.action)) naresid(object$.na.action, object$residuals)
     else object$residuals
 }
```

8 Matrix conversions

The package currently uses objects from both the *Matrix* and the *bdsmatrix* libraries. The former are the basic unit for the `lmekin` function, and are returned as kinship matrices by the *kinship2* library. Bsdmatrix objects are the main tool for the internal routines of `coxme`, although there is a long term goal of changing that in order to gain more flexibility.

In the meantime, we need programs to convert from one to the other. The major nuisance is that the sparse portion of a bdsmatrix object is stored in row-major order, equivalent to an upper triangular dsRMatrix object in the Matrix library. However, the routines we need to use for QR decompostions are supported for column-major sparse objects. Our first routine, therefore, is one for rearrangement of a sparse block. Consider a 4 by 4 sparse in bdsmatrix order

$$\begin{pmatrix} 1 & & & \\ 2 & 5 & & \\ 3 & 6 & 8 & \\ 4 & 7 & 9 & 10 \end{pmatrix}$$

A dsCMatrix object expects the order 1, 2,5, 3,6,8, 4,7,9,10; read across the rows rather than down. In the other direction we want the inverse of this, namely 1, 2, 4, 7, 3, 5, 8, 6, 9, 10;

which are the positions of 1, 2, 3, ... in the first list. (Conversely, our first list is the positions of 1, 2, 3, ... in this list.)

⟨*bdsmatrix*⟩=
```
 #Functions for moving back and forth between Matrix and bdsmatrix objects
 rowTocol <- function(bs) {  #bs = size of block
     n <- (bs*(bs+1))/2
     indx <- integer(n)
     offset <- c(0L, cumsum(seq.int(bs-1, 1)))
     k <- 1L
     for (i in seq.int(1,bs)) {
         for (j in 1:i ) {
             indx[k] <- i + offset[j]
             k <- k+1
             }
         }
     indx
 }

 colTorow <- function(bs) {  #bs = size of block
     n <- (bs*(bs+1))/2
     indx <- integer(n)
     offset <- c(0L, cumsum(seq.int(1, bs-1)))
     k <- 1L
     for (i in seq.int(1,bs)) {
         for (j in seq.int(i, bs) ) {
             indx[k] <- i + offset[j]
             k <- k+1
             }
         }
     indx
 }
```

Now for the actual conversion, which is mostly a bookkeeping/counting operation. The majority of the work is creating the indices for the second object. In the rare case that the bdsmatrix object has no sparse portion we can convert the rmat portion directly using the `symmpart` function. Otherwise we use a C function, after finding that this conversion process was the major component of the run time for lmekin: it happens once per iteration whereas conversions the other way only occur a single time. The key of the algorithm is to note the pattern of numbers above: the indices for the diagonal are 0, 2, 2+3, 2+3+4, ..., (remember that the C indices start at zero), and the difference between rows is 1, 2, 3, ... A bdsmatrix allows dimnames to be an integer, but a regular R matrix or Matrix object does not.

⟨*bdsmatrix*⟩=
```
 setAs("bdsmatrix", "dsCMatrix", function(from) {
     if (length(from@blocks)==0) symmpart(Matrix(from@rmat))
     else {
```

```
        temp <- .Call(Cbds_dsc,
                      from@blocksize,
                      from@blocks,
                      from@rmat,
                      from@Dim)
         new("dsCMatrix",
                 i = temp$i,
                 p = temp$p,
                 x=  temp$x,
                 Dim = from@Dim,
                 Dimnames= lapply(from@Dimnames, as.character),
                 uplo='U',
                 factors=list())
    }
 })
```

⟨*bds_dsc*⟩=

```
 #include "coxmeS.h"
 SEXP bds_dsc(SEXP blocksize2,  SEXP blocks2,  SEXP rmat2,
              SEXP dim2) {
     int i,j, iblock, bstart;
     int n, k, k2, rsize;
     int bs, nblock, rcol;

     /* pointers to input arguments */
     int *blocksize, dim;
     double *blocks, *rmat;

     /* output arguments */
     SEXP retlist, reti2, retp2, retx2;
     int *reti, *retp;
     double *retx;
     static const char *outnames[] = {"i", "p", "x", ""};

     /* Get sizes */
     blocksize = INTEGER(blocksize2);
     blocks = REAL(blocks2);
     rmat =  REAL(rmat2);
     dim = (INTEGER(dim2))[0];
     rcol = ncols(rmat2);

     nblock = LENGTH(blocksize2);  /* number of blocks */
     n = LENGTH(blocks2);          /* total number of non-zero elements */
     rsize = rcol*dim - (rcol*(rcol-1))/2;  /* the dense part */

     /* create output objects */
```

```
PROTECT(reti2 = allocVector(INTSXP, n + rsize));
reti = INTEGER(reti2);
PROTECT(retp2 = allocVector(INTSXP, dim+1));
retp = INTEGER(retp2);
PROTECT(retx2 = allocVector(REALSXP, n + rsize));
retx = REAL(retx2);

k=0;  /* total elements processed */
bstart =0;  /* row number for start of block */
*retp =0;
for (iblock=0; iblock<nblock; iblock++) {
    bs = blocksize[iblock];
    for (i=0; i<bs; i++) {   /* column in the output */
        retp[1] = *retp + i + 1; retp++;
        k2 = i+k;
        for (j=0; j<=i; j++) { /* row in the output*/
             *retx++ = blocks[k2];
            *reti++ = bstart + j;
            k2 += bs - (j+1);
            }
        }
    bstart += bs;
    k += bs * (bs+1)/2;
    }

/* Now do the rmat portion, if present
   But not the lower right corner of rmat
*/
k = 1+ dim - rcol;
for (i=0; i<rcol; i++) {
    retp[1] = *retp + k; retp++;
    for (j=0; j<k; j++) {
        *retx++ = rmat[j];
        *reti++ = j;
        }
    rmat += dim;
    k++;
    }

retlist = PROTECT(mkNamed(VECSXP, outnames));
SET_VECTOR_ELT(retlist, 0, reti2);
SET_VECTOR_ELT(retlist, 1, retp2);
SET_VECTOR_ELT(retlist, 2, retx2);
UNPROTECT(4);
return(retlist);
}
```

Conversion of the result of a cholesky decompostion leads to the same matrix form. However, the dtCMatrix object is an L'L decompostion, not an LDL' one, so we have to multiply things out. the same.

⟨*bdsmatrix*⟩=

```
 setAs("gchol.bdsmatrix", "dtCMatrix", function(from) {
     dd <- sqrt(diag(from))  #the multiplication factor
     rownum <- function(z) unlist(lapply(1:z, function(r) 1:r))
     nb <- from@blocksize* (from@blocksize+1)/2  #elements per block
     if (length(from@blocks)>0){
         temp <- vector('list', length(nb))
         bstart <- c(0, cumsum(from@blocksize)) #offset of each block
         for (i in 1:length(nb))
             temp[[i]] <- rownum(from@blocksize[i]) + bstart[i]
         m.i <- unlist(temp)
         m.p <- unlist(lapply(from@blocksize, function(x) seq(1,x)))

         xindx <- unlist(sapply(from@blocksize, rowTocol)) +
                  rep.int(c(0, cumsum(nb))[1:length(nb)], nb)
         m.x <- from@blocks[xindx]

         if (length(from@rmat >0)) {
             nc <- ncol(from@rmat)  #number of columns in rmat
             nr <- nrow(from)     #total number of rows
             ii <- seq(to=nr, length=nc)
             m.i <- c(m.i, unlist(lapply(ii, function(r) 1:r)))
             m.p <- c(m.p, ii)
             m.x <- c(m.x, from@rmat[row(from@rmat) <= nr + col(from@rmat) -nc])
         }
     }
     else {
         nc <- ncol(from@rmat)  #number of columns in rmat
         nr <- nrow(from)     #total number of rows
         ii <- seq(to=nr, length=nc)
         m.i <- unlist(lapply(ii, function(r) 1:r))
         m.p <- ii
         m.x <- from@rmat[row(from@rmat) <= nr + col(from@rmat) -nc]
     }

     #Modify x
     m.x <- m.x * dd[m.i]  # fixes the off diagonals
     m.x[rep(1:length(m.p), m.p) ==m.i] <- dd  #diagonals
     new("dtCMatrix",
         i = as.integer(m.i-1),
         p = as.integer(c(0, cumsum(m.p))),
         Dim= dim(from),
```

```
        Dimnames=from@Dimnames,
        x= m.x,
        uplo='U',
        diag='N')
})
```

If someone is using the original kinship library then kinship matrices will be a bdsmatrix object, if they are using kinship2 they will be Matrix objects. For now, we want to turn Matrix objects into bdsmatrix ones. Doing so in the most general way is not trivial since that would involve recognizing a best rmat portion. We simply find connected blocks, which will work for kinship matrices.

Our first job is to recognize a block. A dsCMatrix stores the upper triangle of the matrix, so for any column it is easy to see the minimal row index with a non-zero value. Imagine starting at the lower left corner, then move to the left keeping track of the lowest row number seen so far. Suppose at column k the min so far is also k. Then we know that the lower k by k block must have only zeros above it, and by symmetry only zeros to the left. Set it aside and start over. We see that any column for which the minimal index for all columns to the right = the current column number will be the start of a block. Assuming that the row indices are in increasing order (I have not yet seen an exception), then `x@i[x@p[1:ncol ]]` will be the index of these minimal elements, using the 0 based indexing of Matrix objects.

Given a block, note that we can't use the simple colTorow function above to rearrange its contents; the Matrix object will have "holes" in it. For example, in a simple family of 2 founders and their 3 children the bdsmatrix object will be a 5 by 5 block with a zero for the pair of founders; the Matrix object from kinship2 will supress the zero. The natrual thing is to use x[i,i] where i is the integer vector delimiting the block. However, at this time Matrix (version 1.0-1) has a major memory leak when subscripting a sparse matrix. The work around is the `getblock` function below. It can very quickly extract the relevant portion of the dsCMatrix object, under the assumption that the matrix is block diagonal and we are extracting an entire block. Because of marry-ins to a pedigree the case where start=end is quite common so we treat it as a special case.

⟨*bdsmatrix*⟩=
```
 # Code to find the subset myself
 #  This ONLY works for the special case below
 getblock <- function(x, start, end) {
     nrow <-  as.integer(1+end-start)
     xp <- x@p[start:(end+1)]
     if (nrow==1) return(x@x[xp[1]+1])  #singleton element

     keep <- (1+min(xp)):max(xp)
     new("dsCMatrix", i=x@i[keep]+ 1L - as.integer(start),
         p= xp- min(xp),
         Dim=c(nrow, nrow), Dimnames=list(NULL, NULL),
         x = x@x[keep], uplo=x@uplo, factors=list())
 }

 setAs("dsCMatrix", "bdsmatrix", function(from) {
```

```
    dd <- dim(from)
    if (dd[1] != dd[2]) stop("Variance matrices must be square")

    nc <- ncol(from)
    minrow <- from@i[from@p[1:nc] +1] +1
    minrow <- rev(cummin(rev(minrow)))
    block.start <- which(1:nc == minrow)
    block.end <- c(block.start[-1] -1, ncol(from))
    nblock <- length(block.start)

    blocks <- vector('list', nblock)
    for (i in 1:nblock) {
#        indx <- block.start[i]:block.end[i]
#        blocks[[i]] <- as.matrix(from[indx, indx])
        blocks[[i]] <- as.matrix(getblock(from, block.start[i], block.end[i]))
    }
    bdsmatrix(blocksize=sapply(blocks, nrow), blocks=unlist(blocks),
              dimnames=dimnames(from))
    })
```