pdInd: G matrix with patterns of zeros

Georges Monette

2018-04-26

`pdInd` is a constructor for `pdClasses` that define G matrices to model the variance of random effects for models in the `nlme` package.

Mixed models in which many predictors have random slopes often fail to converge in part because of the large number of parameters in the full covariance (G) matrix for random effects. One way of fitting a more parsimonious model that includes random slopes is to use `pdDiag` with zeros off the diagonal. However, this also forces zero covariances between random slopes and and the random intercept, resulting in a model that is not equivariant with respect to location transformations of the predictors with random slopes. The alternative remedy of omitting random slopes for some predictors can lead to biased estimates and incorrect standard errors of regression coefficients.

The default covariance pattern for `pdInd` produces a G matrix with zero covariances except in the first row and column. If the first random effect is the intercept, the resulting model assumes independence between random slopes without imposing minimality of variance over the possibly arbitrary origin. This imposition is the reason that having all covariances equal to zero results in a model that fails to be equivariant under location transformations.

The optional `cov` parameter can be used to allow selected non-zero covariance between random slopes.

For example, if two variables, X1 and X2 have random effects, the random effects model would be specified in a call to `lme` as `random = ~ 1 + X1 + X2`.

The default G matrix has the form:

$$G = \begin{pmatrix} g_{00} & g_{01} & g_{02} \\ g_{10} & g_{11} & g_{12} \\ g_{20} & g_{21} & g_{22} \end{pmatrix}$$

With `pdDiag`, all the off-diagonal elements of G are constrained to 0. Forcing g_{01} and g_{02} to be 0 produces a model that is not equivariant with respect to location changes in `X1` and `X2`. The value at which the variance of `Y` given `X1` and `X2` is minimized is forced to be 0 for both variables.

However, constraining $g_{12} = 0$ produces a model that is equivariant with respect to location-scale transformation of `X1` and `X2` and in which the random between cluster values of regression slopes for each variable are independent of each other.

The `pdInd` class of positive-definite matrices creates, by default, a matrix with arbitrary values along the diagonal and in the first row and column, but zeros elsewhere. In the case of a 4×4 matrix, this produces:

$$G = \begin{pmatrix} g_{00} & g_{01} & g_{02} & g_{03} \\ g_{10} & g_{11} & 0 & 0 \\ g_{20} & 0 & g_{22} & 0 \\ g_{30} & 0 & 0 & g_{33} \end{pmatrix}$$

The challenge in parametrizing the G matrix is finding an unconstrained parametrization that results in a positive-definite matrix with selected covariances constrained to 0.

We consider a right-Cholesky decomposition noting that the diagonal component in the following factorization results in a diagonal component in the variance matrix.

$$\begin{pmatrix} G_{11} & G_{12} \\ G_{12}' & G_{22} \end{pmatrix} = \begin{pmatrix} R_{11} & R_{12} \\ 0 & D_{22} \end{pmatrix} \begin{pmatrix} R_{11}' & 0 \\ R_{12}' & D_{22} \end{pmatrix} = \begin{pmatrix} R_{11}R_{11}' + R_{12}R_{12}' & R_{12}D_{22} \\ D_{22}R_{12}' & D_{22}^2 \end{pmatrix}$$

With R_{11} square upper-triangular and D_{22} square diagonal, G_{22} must be diagonal. In addition, patterns of zeros in the R_{12} matrix, lying above the diagonal D_{22} matrix, are preserved in G_{12}.

Note that the unconstrained Cholesky parametrization uses the log of the diagonal elements of the triangular factor.

```
(fac <- cbind( c(1,0,0,0,0), c(1,2,0,0,0), c(0,1,3,0,0), c(1,0,0,4,0), c(0,1,0, 0, 5) ))
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    0    1    0
[2,]    0    2    1    0    1
[3,]    0    0    3    0    0
[4,]    0    0    0    4    0
[5,]    0    0    0    0    5
```

```
fac %*% t(fac)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    3    2    0    4    0
[2,]    2    6    3    0    5
[3,]    0    3    9    0    0
[4,]    4    0    0   16    0
[5,]    0    5    0    0   25
```

Note how the pattern of zeros in the last three columns of the first two rows is preserved in the cross product since the lower 3×3 diagonal block matrix is itself diagonal.

However, if the lower 3×3 block diagonal matrix is not diagonal, then the pattern of zeros in the top two rows above it is not necessarily preserved.

```
(fac <- cbind( c(1,0,0,0,0), c(1,2,0,0,0), c(0,1,3,0,0), c(1,0,0,4,0), c(0,1,0, -1, 5) ))
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    0    1    0
[2,]    0    2    1    0    1
[3,]    0    0    3    0    0
[4,]    0    0    0    4   -1
[5,]    0    0    0    0    5
```

```
fac %*% t(fac)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    3    2    0    4    0
[2,]    2    6    3   -1    5
[3,]    0    3    9    0    0
[4,]    4   -1    0   17   -5
[5,]    0    5    0   -5   25
```

But patterns of zeros above non-diagonal blocks are not necessarily preserved.

The `cov` parameter allows the user to specify a pattern of zeros in the upper triangle of the upper-triangular 'R' factor of the the G matrix. As observed above, in some cases this will result in the same pattern in the G matrix. Even if the pattern in the R factor does not create a similar pattern in the G matrix, the model for the G matrix will nevertheless have the number of additional parameters for covariance as given by the `TRUE` entries in the upper diagonal of the `cov` matrix.

Predictor transformations to improve convergence

We will show how to use `chol(getG(fit))` to suggest transformations of predictors that appear in random effects model to help improve the convergence of mixed models fits.

We conjecture that the condition number of the Hessian matrix for the parameters in the G matrix may be approximately the square of the condition number for the G matrix. Thus a condition number for G in the vicinity of 10^7 would effectively result in singularity of the Hessian.

Rescaling and relocating the predictors with random slopes can greatly improve ill-conditioning of the G matrix.

Let

$$\mathbf{Z} = \begin{pmatrix} 1 \\ Z_1 \\ \vdots \\ Z_k \end{pmatrix}$$

represent the vector of variables with random effects $\mathbf{u}$. For the ith cluster the contribution from level-2 random effects is:

$$\mathbf{Z}_i'\mathbf{u}_i$$

Consider a location-scale tranformation of the variables in $\mathbf{Z}$. It has the form $\mathbf{Z}^* = T\mathbf{Z}$ where T is upper triangular with the form

$$T = \begin{bmatrix} 1 & a_1\, a_2\, \cdots\, a_k \\ 0 & B \end{bmatrix}$$

with B a diagonal matrix containing scaling coefficients, $b_1, b_2, ..., b_k$, so that

$$Z_i^* = a_i + b_i Z_i$$

If $G = Var(\mathbf{u})$ then

$$G^* = T'^{-1} G T^{-1}$$

where $G^* = \text{Var}(\mathbf{u}^*)$ with

$$\mathbf{Z}_i'\mathbf{u}_i = \mathbf{Z}_i^{*\prime}\mathbf{u}_i^*$$

Thefore if we factor

$$G = T'T$$

the upper triangular matrix $\frac{1}{t_{11}}T$ provides a transformation of $\mathbf{Z}$ that minimizes the condition number of G^*. In R, this is simply `getG(fit) %>% chol %>% {./.[1,1]}`.

Improving convergence – IN PROGRESS

1. Location scale transformations of Z variables. Note: don't need to change to X variables but might like to for purposes of inference.
2. Parsimonious G matrices: Use pdInd and pdDiag.
3. Note that LRTs with anova are likely to be informative to compare nested RE models with singular Hessians provided differences are only in covariance structure, i.e. same non-zero diagonal elements.
4. STUDY: any effect on vcov of changing to equivalent Zs without changing X.

Nuts and Bolts of the G matrix in lme – IN PROGRESS

The initial call to `pdConstruct.pdInd` occurs in the initialization phase of `lme` which call `reStruct` which in turn calls `pdMat` that creates an empty `pdInd` object, sets `value <- numeric(0)` and returns:

`pdConstuct(object, value, form, nam, data)`

pdInd methods

```
methods(class='pdInd')
```

```
   [1] pdConstruct pdFactor    pdMatrix    solve
   see '?methods' for accessing help and source code
```

```
gnew:::pdInd
```

```
   function (value = numeric(0), form = NULL, nam = NULL,
               data = sys.parent(), cov = NULL)
     {
       # unchanged
       object <- numeric(0)
       class(object) <- c("pdInd", "pdMat")
       pdConstruct(object, value, form, nam, data, cov)
     }
   <environment: namespace:gnew>
```

```
gnew:::pdConstruct.pdInd
```

```
   function (object, value = numeric(0), form = formula(object),
               nam = Names(object), data = sys.parent(),
               cov = NULL,
               ...)
     {
       # note that pdConstruct.pdMat return an upper-triangular R factor, === might not be correct
       if(!is.null(attr(object,'cov'))) cov <- attr(object,'cov')
       if(!is.null(attr(value,'cov'))) cov <- attr(value,'cov')
       val <- nlme:::pdConstruct.pdMat(object,
                                       value = value,
                                       form = form,
                                       nam = nam,
                                       data = data)
       attr(val,'cov') <- cov
       if (length(val) == 0) {
         class(val) <- c("pdInd", "pdMat")
         return(val)
       }
       # mod 2015 07 04: added arbitrary cov structure of non zero
       # covariance
       isRmat <- function(x) all( x[row(x) > col(x)] == 0) # is lower triangle == 0?
       if (is.matrix(val)) {
         if(is.null(cov)) {
           if(!is.null(attr(val,'cov'))) cov <- attr(val,'cov')
           else cov <- (row(val) == 1) & (col(val) > 1)
         }
```

```
        #    disp(cov)
        if(isRmat(val)){
          value <- c(log(diag(val)), val[cov])
          # keeping only the entries that should be non-zero
        } else stop("matrix should be an upper triangular matrix")
        attributes(value) <-
          attributes(val)[names(attributes(val)) != "dim"]
        attr(value,"cov") <- cov
        class(value) <- c("pdInd", "pdMat")
        attr(value,"invert") <- FALSE
        return(value)
      }
      stop("shouldn't get here in pdConstruct.pdInd")
      Ncol <- (length(val) + 1)/2
      if (length(val) != 2*round(Ncol) - 1) {
        stop(gettextf("an object of length %d does not match a pdInd factor (diagonal + covariances wi
                      length(val)), domain = NA)
      }
      class(val) <- c("pdInd", "pdMat")
      val
    }
  <environment: namespace:gnew>
```

```
gnew:::pdFactor.pdInd
```

```
  function (object)
    {
      invert <- attr(object,"invert")
      cov <- attr(object,"cov")
      object <- as.vector(object)
      Ncov <- sum(cov)
      Ncol <- length(object) - Ncov
      # was:
      #   L <- matrix(0,Ncol,Ncol)
      #   diag(L) <- exp( object[1:Ncol])
      #   if ( Ncol > 1 ) L[row(L)>1 & col(L)==1] <-
      #     object[(Ncol+1):length(object)]
      #   if(invert) c(t(solve(L))) else c(L2R(L))
      R <- matrix(0,Ncol,Ncol)
      diag(R) <- exp( object[1:Ncol])
      if ( Ncol > 1 ) R[cov] <-
        object[(Ncol+1):length(object)]
      if(invert) c(t(solve(f2L(R)))) else c(R)
    }
  <environment: namespace:gnew>
```

```
gnew:::pdMatrix.pdInd
```

```
  function (object, factor = FALSE)
    {
      if (!isInitialized(object)) {
        stop("cannot extract matrix from an uninitialized object")
      }
      cov <- attr(object,"cov")
      Ncov <- sum(cov)
      Ncol <- length(object) - Ncov
      value <- array(pdFactor(object), c(Ncol, Ncol),
                     attr(object, "Dimnames"))
      ob <- as.vector(object) # subsetting object calls pdMatrix!
      attr(value, "logDet") <- 2*sum(ob[1:Ncol])
      if (factor) value else  crossprod(value)
    }
  <environment: namespace:gnew>
```

```r
gnew:::solve.pdInd
```

```
function (a, b, ...)
  {
    if (!isInitialized(a)) {
      stop("cannot get the inverse of an uninitialized object")
    }
    attr(a, 'invert') <- !attr(a, 'invert')
    a
    #    Ncol <- (length(a) + 1)/2
    #    ob <- as.vector(a)
    #    if( Ncol == 1) ret <- -ob[1]
    #    else ret <-
    #      c( -ob[1:Ncol] ,
    #        - exp(ob[1])*ob[(Ncol+1):length(ob)]/exp(ob[2:Ncol]))
    #    attributes(ret) <- attributes(a)
    #    ret
  }
<environment: namespace:gnew>
```

pdMat methods

```r
methods(class='pdMat')
```

```
 [1] [              [<-            as.matrix      coef           coef<-
 [6] corMatrix      Dim            formula        isInitialized logDet
[11] matrix<-       Names          Names<-        pdConstruct    pdFactor
[16] pdMatrix       plot           print          solve          summary
[21] VarCorr
see '?methods' for accessing help and source code
```

```r
nlme:::pdMat
```

```
function (value = numeric(0), form = NULL, nam = NULL, data = sys.frame(sys.parent()),
    pdClass = "pdSymm")
{
    if (inherits(value, "pdMat")) {
        pdClass <- class(value)
    }
    object <- numeric(0)
    class(object) <- unique(c(pdClass, "pdMat"))
    pdConstruct(object, value, form, nam, data)
}
<bytecode: 0x000000001f65a960>
<environment: namespace:nlme>
```

```
nlme:::pdConstruct.pdMat
```

```
function (object, value = numeric(0), form = formula(object),
    nam = Names(object), data = sys.frame(sys.parent()), ...)
{
    if (inherits(value, "pdMat")) {
        if (length(form) == 0) {
            form <- formula(value)
        }
        if (length(nam) == 0) {
            nam <- Names(value)
        }
        if (isInitialized(value)) {
            return(pdConstruct(object, as.matrix(value), form,
                nam, data))
        }
        else {
            return(pdConstruct(object, form = form, nam = nam,
                data = data))
        }
    }
    if (length(value) > 0) {
        if (inherits(value, "formula") || data.class(value) ==
            "call") {
            if (!is.null(form)) {
                warning("ignoring argument 'form'")
            }
            form <- formula(value)
            if (length(form) == 3) {
                form <- list(form)
            }
        }
        else if (is.character(value)) {
            if (length(nam) > 0) {
                warning("ignoring argument 'nam'")
            }
            nam <- value
        }
        else if (is.matrix(value)) {
            vdim <- dim(value)
            if (length(vdim) != 2 || diff(vdim) != 0) {
                stop("'value' must be a square matrix")
            }
            if (length(unlist(vnam <- dimnames(value))) > 0) {
                vnam <- unique(unlist(vnam))
                if (length(vnam) != vdim[1]) {
                  stop("dimnames of 'value' must match or be NULL")
                }
                dimnames(value) <- list(vnam, vnam)
                if (length(nam) > 0) {
                  if (any(is.na(match(nam, vnam))) || any(is.na(match(vnam,
                    nam)))) {
                    stop("names of 'value' are not consistent with 'nam' argument")
                  }
                value <- value[nam, nam, drop = FALSE]
              }
              else {
                nam <- vnam
              }
            }
            form <- form
            nam <- nam
            object <- chol((value + t(value))/2)
            attr(object, "dimnames") <- NULL
            attr(object, "rank") <- NULL
        }
        else if (is.numeric(value)) {
            value <- as.numeric(value)
            attributes(value) <- attributes(object)
            object <- value
        }
        else if (data.class(value) == "list") {
            if (!is.null(form)) {
                warning("ignoring argument 'form'")
            }
            form <- value
        }
        else {
            stop(gettextf("%s is not a valid object for \"pdMat\"",
                sQuote(deparse(object))), domain = NA)
        }
    }
    if (!is.null(form)) {
        if (inherits(form, "formula") && length(form) == 3) {
            form <- list(form)
        }
        if (is.list(form)) {
            if (any(!unlist(lapply(form, function(el) {
                inherits(el, "formula") && length(el) == 3
            })))) {
                stop("all elements of 'form' list must be two-sided formulas")
            }
            val <- list()
            for (i in seq_along(form)) {
                if (is.name(form[[i]][[2]])) {
                  val <- c(val, list(form[[i]]))
                }
                else {
                  val <- c(val, eval(parse(text = paste("list(",
                    paste(paste(all.vars(form[[i]][[2]]), deparse(form[[i]][[3]]),
                      sep = "~"), collapse = ","), ")"))))
                }
            }
            form <- val
            class(form) <- "listForm"
            namesForm <- Names(form, data)
        }
        else {
            if (inherits(form, "formula")) {
                namesForm <- Names(asOneSidedFormula(form), data)
            }
            else {
                stop("'form' can only be a formula or a list of formulae")
            }
        }
        if (length(namesForm) > 0) {
            if (length(nam) == 0) {
                nam <- namesForm
            }
            else {
                if (any(noMatch <- is.na(match(nam, namesForm)))) {
                  err <- TRUE
                  namCopy <- nam
                  indNoMatch <- seq_along(nam)[noMatch]
                  if (any(wch1 <- (nchar(nam, "c") > 12))) {
                    wch1 <- substring(nam, nchar(nam, "c") -
                      10) == "(Intercept)"
                    if (any(wch1)) {
                      namCopy[indNoMatch[wch1]] <- substring(nam[wch1],
                        1, nchar(nam[wch1], "c") - 12)
                      noMatch[wch1] <- FALSE
                      indNoMatch <- indNoMatch[!wch1]
                    }
                  }
                  if (sum(noMatch) > 0) {
                    namCopy[indNoMatch] <- paste(namCopy[indNoMatch],
                      "(Intercept)", sep = ".")
                  }
                  if (!any(is.na(match(namCopy, namesForm)))) {
                    err <- FALSE
                  }
                  if (err)
                    stop("'form' not consistent with 'nam'")
                }
            }
        }
    }
    if (is.matrix(object)) {
        if (length(nam) > 0 && (length(nam) != dim(object)[2])) {
            stop("length of 'nam' not consistent with dimensions of initial value")
        }
    }
    attr(object, "formula") <- form
    attr(object, "Dimnames") <- list(nam, nam)
    object
}
<bytecode: 0x000000001f28ee88>
<environment: namespace:nlme>
```

```
nlme:::pdMatrix.pdMat
```

```
function (object, factor = FALSE)
{
    if (!isInitialized(object)) {
        stop("cannot access the matrix of uninitialized objects")
    }
    if (factor) {
        stop("no default method for extracting the square root of a \"pdMat\" object")
    }
    else {
        crossprod(pdMatrix(object, factor = TRUE))
    }
}
<bytecode: 0x000000001f4d3738>
<environment: namespace:nlme>
```

```
nlme:::pdFactor.pdMat
```

```
    function (object)
    {
        c(qr.R(qr(pdMatrix(object))))
    }
    <bytecode: 0x000000001f475828>
    <environment: namespace:nlme>
```

```
nlme:::solve.pdMat
```

```
    function (a, b, ...)
    {
        if (!isInitialized(a)) {
            stop("cannot get the inverse of an uninitialized object")
        }
        matrix(a) <- solve(as.matrix(a))
        a
    }
    <bytecode: 0x000000001f3ead50>
    <environment: namespace:nlme>
```

```
nlme:::VarCorr.pdMat
```

```
    function (x, sigma = 1, rdig = 3, ...)
    {
        sx <- summary(x)
        sd <- sigma * attr(sx, "stdDev")
        var <- sd^2
        p <- dim(sx)[2]
        v <- array(c(var, sd), c(p, 2), list(names(sd), c("Variance",
            "StdDev")))
        attr(v, "formStr") <- if (inherits(attr(x, "formula"), "listForm")) {
            paste(class(x)[[1]], "(list(", paste(sapply(attr(x, "formula"),
                function(x) as.character(deparse(x))), collapse = ","),
                "))", sep = "")
        }
        else {
            paste(class(x)[[1]], "(", substring(deparse(attr(x, "formula")),
                2), ")", sep = "")
        }
        if (attr(sx, "noCorrelation") || p <= 1)
            return(v)
        ll <- lower.tri(sx)
        sx[ll] <- format(round(sx[ll], digits = rdig))
        sx[!ll] <- ""
        if (!is.null(colnames(sx))) {
            sx[1, ] <- abbreviate(colnames(sx), minlength = rdig +
                3)
        }
        dimnames(sx) <- list(names(sd), c("Corr", rep("", p - 1)))
        attr(v, "corr") <- sx[, -p, drop = FALSE]
        v
    }
    <bytecode: 0x000000001f630af0>
    <environment: namespace:nlme>
```

```
nlme:::as.matrix.pdMat
```

```
    function (x, ...)
    pdMatrix(x)
    <bytecode: 0x000000001f1cb278>
    <environment: namespace:nlme>
```

```
nlme:::`matrix<-.pdMat`
```

```
    function (object, value)
    {
        value <- as.matrix(value)
        if (isInitialized(object) && any(dim(value) != Dim(object))) {
            stop("cannot change dimensions on an initialized \"pdMat\" object")
        }
        pdConstruct(object, value)
    }
    <bytecode: 0x000000001f161e08>
    <environment: namespace:nlme>
```

```
nlme:::coef.pdMat
```

```
    function (object, unconstrained = TRUE, ...)
    {
        if (unconstrained || !isInitialized(object)) {
            as.vector(object)
        }
        else {
            stop("do not know how to obtain constrained coefficients")
        }
    }
    <bytecode: 0x000000001f0f8100>
    <environment: namespace:nlme>
```

```
nlme:::`coef<-.pdMat`
```

```
    function (object, ..., value)
    {
        value <- as.numeric(value)
        if (isInitialized(object)) {
            if (length(value) != length(object)) {
                stop("cannot change the length of the parameter after initialization")
            }
        }
        else {
            return(pdConstruct(object, value))
        }
        class(value) <- class(object)
        attributes(value) <- attributes(object)
        value
    }
    <bytecode: 0x000000001f08be08>
    <environment: namespace:nlme>
```

pdSymm methods

```
nlme:::pdSymm
```

```
function (value = numeric(0), form = NULL, nam = NULL, data = parent.frame())
{
    object <- numeric(0)
    class(object) <- c("pdSymm", "pdMat")
    pdConstruct(object, value, form, nam, data)
}
<bytecode: 0x000000001c56b638>
<environment: namespace:nlme>
```

```
methods(class='pdSymm')
```

```
[1] coef        Dim         logDet      pdConstruct pdFactor    pdMatrix
[7] solve       summary
see '?methods' for accessing help and source code
```

```
nlme:::pdConstruct.pdSymm
```

```
function (object, value = numeric(0), form = formula(object),
    nam = Names(object), data = sys.frame(sys.parent()), ...)
{
    val <- NextMethod()
    if (length(val) == 0) {
        class(val) <- c("pdSymm", "pdMat")
        return(val)
    }
    if (is.matrix(val)) {
        vald <- svd(val, nu = 0)
        object <- vald$v %*% (log(vald$d) * t(vald$v))
        value <- object[row(object) <= col(object)]
        attributes(value) <- attributes(val)[names(attributes(val)) !=
            "dim"]
        class(value) <- c("pdSymm", "pdMat")
        return(value)
    }
    Ncol <- round((sqrt(8 * length(val) + 1) - 1)/2)
    if (length(val) != round((Ncol * (Ncol + 1))/2)) {
        stop(gettextf("an object of length %d does not match the required parameter size",
            length(val)), domain = NA)
    }
    class(val) <- c("pdSymm", "pdMat")
    val
}
<bytecode: 0x000000001b656e30>
<environment: namespace:nlme>
```

```
nlme:::pdMatrix.pdSymm
```

```
function (object, factor = FALSE)
{
    if (!isInitialized(object))
        stop("cannot extract matrix from an uninitialized object")
    if (factor) {
        Ncol <- Dim(object)[2]
        value <- array(pdFactor(object), c(Ncol, Ncol), attr(object,
            "Dimnames"))
        attr(value, "logDet") <- sum(log(abs(svd.d(value))))
        value
    }
    else {
        NextMethod()
    }
}
<bytecode: 0x00000000239c4d98>
<environment: namespace:nlme>
```

```
nlme:::pdFactor.pdSymm
```

```
function (object)
{
    Ncol <- round((-1 + sqrt(1 + 8 * length(object)))/2)
    .C(matrixLog_pd, Factor = double(Ncol * Ncol), as.integer(Ncol),
        as.double(object))$Factor
}
<bytecode: 0x000000002388d240>
<environment: namespace:nlme>
```

```
nlme:::solve.pdSymm
```

```
function (a, b, ...)
{
    if (!isInitialized(a)) {
        stop("cannot extract the inverse from an uninitialized object")
    }
    coef(a) <- -coef(a, TRUE)
    a
}
<bytecode: 0x0000000023a61f18>
<environment: namespace:nlme>
```

```
nlme:::coef.pdSymm
```

```
function (object, unconstrained = TRUE, ...)
{
    if (unconstrained || !isInitialized(object))
        NextMethod()
    else {
        val <- as.matrix(object)
        aN <- Names(object)
        aN1 <- paste("cov(", aN, sep = "")
        aN2 <- paste(aN, ")", sep = "")
        aNmat <- t(outer(aN1, aN2, paste, sep = ","))
        aNmat[row(aNmat) == col(aNmat)] <- paste("var(", aN,
            ")", sep = "")
        val <- val[row(val) <= col(val)]
        names(val) <- aNmat[row(aNmat) <= col(aNmat)]
        val
    }
}
<bytecode: 0x0000000023adc700>
<environment: namespace:nlme>
```

lme

```
nlme:::lme.formula
```

```
function (fixed, data = sys.frame(sys.parent()), random = pdSymm(eval(as.call(fixed[-2]))),
    correlation = NULL, weights = NULL, subset, method = c("REML",
        "ML"), na.action = na.fail, control = list(), contrasts = NULL,
    keep.data = TRUE)
{
    Call <- match.call()
    miss.data <- missing(data) || !is.data.frame(data)
    controlvals <- lmeControl()
    if (!missing(control)) {
        controlvals[names(control)] <- control
    }
    fixedSigma <- controlvals$sigma > 0
    if (!inherits(fixed, "formula") || length(fixed) != 3) {
        stop("\nfixed-effects model must be a formula of the form \"resp ~ pred\"")
    }
    method <- match.arg(method)
    REML <- method == "REML"
    reSt <- reStruct(random, REML = REML, data = NULL)
    groups <- getGroupsFormula(reSt)
    if (is.null(groups)) {
        if (inherits(data, "groupedData")) {
            groups <- getGroupsFormula(data)
            namGrp <- rev(names(getGroupsFormula(data, asList = TRUE)))
            Q <- length(namGrp)
            if (length(reSt) != Q) {
                if (length(reSt) != 1) {
                  stop("incompatible lengths for 'random' and grouping factors")
                }
                randL <- vector("list", Q)
                names(randL) <- rev(namGrp)
                for (i in 1:Q) randL[[i]] <- random
                reSt <- reStruct(as.list(randL), REML = REML,
                  data = NULL)
            }
            else {
                names(reSt) <- namGrp
            }
        }
        else {
            groups <- ~1
            names(reSt) <- "1"
        }
    }
    if (!is.null(correlation)) {
        add.form <- FALSE
        if (!is.null(corGrpsForm <- getGroupsFormula(correlation,
            asList = TRUE))) {
            corGrpsForm <- unlist(lapply(corGrpsForm, function(el) deparse(el[[2L]])))
            lmeGrpsForm <- unlist(lapply(splitFormula(groups),
                function(el) deparse(el[[2L]])))
            corQ <- length(corGrpsForm)
            lmeQ <- length(lmeGrpsForm)
            if (corQ <= lmeQ) {
                if (any(corGrpsForm != lmeGrpsForm[1:corQ])) {
                  stop("incompatible formulas for groups in 'random' and 'correlation'")
                }
                if (corQ < lmeQ) {
                  warning("cannot use smaller level of grouping for 'correlation' than for 'random'.
                  add.form <- TRUE
                }
            }
            else if (any(lmeGrpsForm != corGrpsForm[1:lmeQ])) {
                stop("incompatible formulas for groups in 'random' and 'correlation'")
            }
        }
        else {
            add.form <- TRUE
            corQ <- lmeQ <- 1
        }
        if (add.form)
            attr(correlation, "formula") <- eval(substitute(~COV |
                GRP, list(COV = getCovariateFormula(formula(correlation))[[2L]],
                GRP = groups[[2L]])))
    }
    else {
        corQ <- lmeQ <- 1
    }
    lmeSt <- lmeStruct(reStruct = reSt, corStruct = correlation,
        varStruct = varFunc(weights))
    mfArgs <- list(formula = asOneFormula(formula(lmeSt), fixed,
        groups), data = data, na.action = na.action)
    if (!missing(subset)) {
        mfArgs[["subset"]] <- asOneSidedFormula(Call[["subset"]])[[2L]]
    }
    mfArgs$drop.unused.levels <- TRUE
    dataMix <- do.call(model.frame, mfArgs)
    origOrder <- row.names(dataMix)
```

```
for (i in names(contrasts)) contrasts(dataMix[[i]]) = contrasts[[i]]
grps <- getGroups(dataMix, groups)
if (inherits(grps, "factor")) {
    ord <- order(grps)
    grps <- data.frame(grps)
    row.names(grps) <- origOrder
    names(grps) <- as.character(deparse((groups[[2L]])))
}
else {
    ord <- do.call(order, grps)
    for (i in 2:ncol(grps)) {
        grps[, i] <- as.factor(paste(as.character(grps[,
            i - 1]), as.character(grps[, i]), sep = "/"))
    }
}
if (corQ > lmeQ) {
    ord <- do.call(order, getGroups(dataMix, getGroupsFormula(correlation)))
}
grps <- grps[ord, , drop = FALSE]
dataMix <- dataMix[ord, , drop = FALSE]
revOrder <- match(origOrder, row.names(dataMix))
N <- nrow(grps)
Z <- model.matrix(reSt, dataMix)
ncols <- attr(Z, "ncols")
Names(lmeSt$reStruct) <- attr(Z, "nams")
contr <- attr(Z, "contr")
X <- model.frame(fixed, dataMix)
Terms <- attr(X, "terms")
auxContr <- lapply(X, function(el) if (inherits(el, "factor") &&
    length(levels(el)) > 1)
    contrasts(el))
contr <- c(contr, auxContr[is.na(match(names(auxContr), names(contr)))])
contr <- contr[!unlist(lapply(contr, is.null))]
X <- model.matrix(fixed, data = X)
y <- eval(fixed[[2L]], dataMix)
ncols <- c(ncols, dim(X)[2L], 1)
Q <- ncol(grps)
attr(lmeSt, "conLin") <- list(Xy = array(c(Z, X, y), c(N,
    sum(ncols)), list(row.names(dataMix), c(colnames(Z),
    colnames(X), deparse(fixed[[2L]])))), dims = MEdims(grps,
    ncols), logLik = 0, sigma = controlvals$sigma, auxSigma = 0)
tmpDims <- attr(lmeSt, "conLin")$dims
if (max(tmpDims$ZXlen[[1L]]) < tmpDims$qvec[1L]) {
    warning(gettextf("fewer observations than random effects in all level %s groups",
        Q), domain = NA)
}
fixDF <- getFixDF(X, grps, attr(lmeSt, "conLin")$dims$ngrps,
    terms = Terms)
lmeSt <- Initialize(lmeSt, dataMix, grps, control = controlvals)
parMap <- attr(lmeSt, "pmap")
if (length(lmeSt) == 1) {
    oldConLin <- attr(lmeSt, "conLin")
    decomp <- TRUE
    attr(lmeSt, "conLin") <- MEdecomp(attr(lmeSt, "conLin"))
```

```
}
else decomp <- FALSE
numIter <- 0
repeat {
    oldPars <- coef(lmeSt)
    optRes <- if (controlvals$opt == "nlminb") {
        control <- list(iter.max = controlvals$msMaxIter,
            eval.max = controlvals$msMaxEval, trace = controlvals$msVerbose)
        keep <- c("abs.tol", "rel.tol", "x.tol", "xf.tol",
            "step.min", "step.max", "sing.tol", "scale.init",
            "diff.g")
        control <- c(control, controlvals[names(controlvals) %in%
            keep])
        nlminb(c(coef(lmeSt)), function(lmePars) -logLik(lmeSt,
            lmePars), control = control)
    }
    else {
        reltol <- controlvals$reltol
        if (is.null(reltol))
            reltol <- 100 * .Machine$double.eps
        control <- list(trace = controlvals$msVerbose, maxit = controlvals$msMaxIter,
            reltol = if (numIter == 0) controlvals$msTol else reltol)
        keep <- c("fnscale", "parscale", "ndeps", "abstol",
            "alpha", "beta", "gamma", "REPORT", "type", "lmm",
            "factr", "pgtol", "temp", "tmax")
        control <- c(control, controlvals[names(controlvals) %in%
            keep])
        optim(c(coef(lmeSt)), function(lmePars) -logLik(lmeSt,
            lmePars), control = control, method = controlvals$optimMethod)
    }
    coef(lmeSt) <- optRes$par
    attr(lmeSt, "lmeFit") <- MEestimate(lmeSt, grps)
    if (!needUpdate(lmeSt)) {
        if (optRes$convergence) {
            msg <- gettextf("%s problem, convergence error code = %s\n  message = %s",
              controlvals$opt, optRes$convergence, paste(optRes$message,
                collapse = ""))
            if (!controlvals$returnObject)
              stop(msg, domain = NA)
            else warning(msg, domain = NA)
        }
        break
    }
    numIter <- numIter + 1L
    lmeSt <- update(lmeSt, dataMix)
    aConv <- coef(lmeSt)
    conv <- abs((oldPars - aConv)/ifelse(aConv == 0, 1, aConv))
    aConv <- NULL
    for (i in names(lmeSt)) {
        if (any(parMap[, i])) {
            aConv <- c(aConv, max(conv[parMap[, i]]))
            names(aConv)[length(aConv)] <- i
        }
    }
```

```
        if (max(aConv) <= controlvals$tolerance) {
            break
        }
        if (numIter > controlvals$maxIter) {
            msg <- gettext("maximum number of iterations (lmeControl(maxIter)) reached without conve
            if (controlvals$returnObject) {
                warning(msg, domain = NA)
                break
            }
            else stop(msg, domain = NA)
        }
    }
    lmeFit <- attr(lmeSt, "lmeFit")
    names(lmeFit$beta) <- namBeta <- colnames(X)
    attr(fixDF, "varFixFact") <- varFix <- lmeFit$sigma * lmeFit$varFix
    varFix <- crossprod(varFix)
    dimnames(varFix) <- list(namBeta, namBeta)
    Fitted <- fitted(lmeSt, level = 0:Q, conLin = if (decomp)
        oldConLin
    else attr(lmeSt, "conLin"))[revOrder, , drop = FALSE]
    Resid <- y[revOrder] - Fitted
    rownames(Resid) <- rownames(Fitted) <- origOrder
    attr(Resid, "std") <- lmeFit$sigma/(varWeights(lmeSt)[revOrder])
    grps <- grps[revOrder, , drop = FALSE]
    lmeSt$reStruct <- solve(lmeSt$reStruct)
    dims <- attr(lmeSt, "conLin")$dims[c("N", "Q", "qvec", "ngrps",
        "ncol")]
    attr(lmeSt, "fixedSigma") <- fixedSigma
    apVar <- if (controlvals$apVar) {
        lmeApVar(lmeSt, lmeFit$sigma, .relStep = controlvals[[".relStep"]],
            minAbsPar = controlvals[["minAbsParApVar"]], natural = controlvals[["natural"]])
    }
    else {
        "Approximate variance-covariance matrix not available"
    }
    attr(lmeSt, "conLin") <- NULL
    attr(lmeSt, "lmeFit") <- NULL
    grpDta <- inherits(data, "groupedData")
    structure(class = "lme", list(modelStruct = lmeSt, dims = dims,
        contrasts = contr, coefficients = list(fixed = lmeFit$beta,
            random = lmeFit$b), varFix = varFix, sigma = lmeFit$sigma,
        apVar = apVar, logLik = lmeFit$logLik, numIter = if (needUpdate(lmeSt)) numIter,
        groups = grps, call = Call, terms = Terms, method = method,
        fitted = Fitted, residuals = Resid, fixDF = fixDF, na.action = attr(dataMix,
            "na.action"), data = if (keep.data && !miss.data) data),
        units = if (grpDta)
            attr(data, "units"), labels = if (grpDta)
            attr(data, "labels"))
}
<bytecode: 0x0000000023dbb6e0>
<environment: namespace:nlme>
```

```
nlme:::reStruct
```

```
function (object, pdClass = "pdLogChol", REML = FALSE, data = sys.frame(sys.parent()))
```

```
{
    if (inherits(object, "reStruct")) {
        if (!missing(REML))
            attr(object, "settings")[1] <- as.integer(REML)
        object[] <- lapply(object, function(el, data) {
            pdMat(el, data = data)
        }, data = data)
        return(object)
    }
    plen <- NULL
    if (inherits(object, "formula")) {
        if (is.null(grpForm <- getGroupsFormula(object, asList = TRUE))) {
            object <- list(object)
        }
        else {
            if (length(object) == 3) {
                object <- eval(parse(text = paste(deparse(getResponseFormula(object)[[2]]),
                  deparse(getCovariateFormula(object)[[2]], width.cutoff = 500),
                  sep = "~")))
            }
            else {
                object <- getCovariateFormula(object)
            }
            object <- rep(list(object), length(grpForm))
            names(object) <- names(grpForm)
        }
    }
    else if (inherits(object, "pdMat")) {
        if (is.null(formula(object))) {
            stop("\"pdMat\" element must have a formula")
        }
        object <- list(object)
    }
    else {
        if (data.class(object) != "list") {
            stop("'object' must be a list or a formula")
        }
        if (is.null(names(object)) && all(unlist(lapply(object,
            function(el) {
                inherits(el, "formula") && length(el) == 3
            })))) {
            object <- list(object)
        }
        else {
            object <- lapply(object, function(el) {
                if (inherits(el, "pdMat")) {
                  if (is.null(formula(el))) {
                    stop("\"pdMat\" elements must have a formula")
                  }
                  return(el)
                }
                if (inherits(el, "formula")) {
                  grpForm <- getGroupsFormula(el)
                  if (!is.null(grpForm)) {
                    el <- getCovariateFormula(el)
                    attr(el, "grpName") <- deparse(grpForm[[2]])
                  }
                  return(el)
                }
                else {
                  if (data.class(el) == "list" && all(unlist(lapply(el,
                    function(el1) {
                      inherits(el1, "formula") && length(el1) ==
                        3
                    })))) {
                    return(el)
                  }
                  else {
                    stop("elements in 'object' must be formulas or \"pdMat\" objects")
                  }
                }
            })
        }
        if (is.null(namObj <- names(object))) {
            namObj <- rep("", length(object))
        }
        aux <- unlist(lapply(object, function(el) {
            if (inherits(el, "formula") && !is.null(attr(el,
                "grpName"))) {
                attr(el, "grpName")
            }
            else ""
        }))
        auxNam <- namObj == ""
        if (any(auxNam)) {
            namObj[auxNam] <- aux[auxNam]
        }
        names(object) <- namObj
    }
    object <- lapply(object, function(el, pdClass, data) {
        pdMat(el, pdClass = pdClass, data = data)
    }, pdClass = pdClass, data = data)
    object <- rev(object)
    if (all(unlist(lapply(object, isInitialized)))) {
        plen <- unlist(lapply(object, function(el) length(coef(el))))
    }
    pC <- unlist(lapply(object, data.class))
    pC <- match(pC, c("pdSymm", "pdDiag", "pdIdent", "pdCompSymm",
        "pdLogChol"), 0) - 1
    attr(object, "settings") <- c(as.integer(REML), 1, 0, pC)
    attr(object, "plen") <- plen
    class(object) <- "reStruct"
    object
}
<bytecode: 0x0000000023e6dca0>
<environment: namespace:nlme>
```