Survival Package Functions

Terry Therneau

July 8, 2021

Contents

1 Introduction

> 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 is the definition of a coding style called *literate programming*. I first made use of it in the *coxme* library and have become a full convert. For the survival library only selected objects are documented in this way; as I make updates and changes I am slowly converting the source code. The first motivation for this is to make the code easier for me, both to create and to maintain. As to maintinance, I have found that whenver I need to update code I spend a lot of time in the "what was I doing in these x lines?" stage. The code never has enough documentation, even for the author. (The survival library is already better than the majority of packages in R, whose comment level is abysmal. In the pre-noweb source code about 1 line in 6 has a comment, for the noweb document the documentation/code ratio is 2:1.) I also find it helps in creating new code to have the real documentation of intent — formulas with integrals and such — closely integrated. The second motivation is to leave code that is well enough explained that someone else can take it over.

The source code is structured using *noweb*, one of the simpler literate programming environments. The source code files look remakably like Sweave, and the .Rnw mode of emacs works perfectly for them. This is not too surprising since Sweave was also based on noweb. Sweave is not sufficient to process the files, however, since it has a different intention: it is designed to *execute* the code and make the results into a report, while noweb is designed to *explain* the code. We do this using the `noweb` library in R, which contains the `noweave` and `notangle` functions. (It would in theory be fairly simple to extend `knitr` to do this task, which is a topic for further exploration one day. A downside to noweb is that like Sweave it depends on latex, which has an admittedly steep learning curve, and markdown is thus attractive.)

2 Cox Models

2.1 Coxph

The `coxph` routine is the underlying basis for all the models. The source was converted to noweb when adding time-transform terms.

The call starts out with the basic building of a model frame and proceeds from there. The aeqSurv function is used to adjucate near ties in the time variable, numerical precision issues that occur when users base caculations on days/365.25 instead of days.

A cluster term in the model is an exception. The variable mentioned is never part of the formal model, and so it is not kept as part of the saved terms structure.

The analysis for multi-state data is a bit more complex.

- If the formula statement is a list, we preprocess this to find out any potential extra variables, and create a new global formula which will be used to create the data frame.
- In the above case missing value processing needs to be deferred, since some covariates may apply only to select transitions.
- After the data frame is constructed, the transitions matrix can be used to check that all the state names actually exist, construct the cmap matrix, and do missing value removal.

⟨*coxph*⟩=

```
 #tt <- function(x) x
 coxph <- function(formula, data, weights, subset, na.action,
         init, control, ties= c("efron", "breslow", "exact"),
         singular.ok =TRUE,  robust,
         model=FALSE, x=FALSE, y=TRUE,  tt, method=ties,
         id, cluster, istate, statedata, nocenter=c(-1, 0, 1), ...) {

     ties <- match.arg(ties)
     Call <- match.call()
     ## We want to pass any ... args to coxph.control, but not pass things
     ##  like "dats=mydata" where someone just made a typo.  The use of ...
     ##  is simply to allow things like "eps=1e6" with easier typing
     extraArgs <- list(...)
     if (length(extraArgs)) {
         controlargs <- names(formals(coxph.control)) #legal arg names
         indx <- pmatch(names(extraArgs), controlargs, nomatch=0L)
         if (any(indx==0L))
             stop(gettextf("Argument %s not matched",
                           names(extraArgs)[indx==0L]), domain = NA)
     }
     if (missing(control)) control <- coxph.control(...)

     # Move any cluster() term out of the formula, and make it an argument
     #  instead.  This makes everything easier.  But, I can only do that with
     #  a local copy, doing otherwise messes up future use of update() on
```

```
#  the model object for a user stuck in "+ cluster()" mode.
if (missing(formula)) stop("a formula argument is required")

ss <- c("cluster", "offset")
if (is.list(formula))
    Terms <- if (missing(data)) terms(formula[[1]], specials=ss) else
             terms(formula[[1]], specials=ss, data=data)
else Terms <- if (missing(data)) terms(formula, specials=ss) else
             terms(formula, specials=ss, data=data)
attr(Terms,'term.labels') = gsub('\n',' ', attr(Terms,'term.labels'))
colnames(attr(Terms,'factors')) = gsub('\n',' ', colnames(attr(Terms,'factors')))
rownames(attr(Terms,'factors')) = gsub('\n',' ', rownames(attr(Terms,'factors')))

tcl <- attr(Terms, 'specials')$cluster
if (length(tcl) > 1) stop("a formula cannot have multiple cluster terms")

if (length(tcl) > 0) { # there is one
    # subscripting of formulas is broken at least through R 3.5, if the
    #  formula contains an offset.  Adding offset to the "specials" above
    #  is just a sneaky way to find out if one is present, then call
    #  reformulate ourselves.  tt is a correct index into the row labels
    #  of the factors attribute, tt+1 to the variables attribute (which is
    #  a list, so you have to skip the "list" call).  The term.labels attr
    #  contains neither the response nor the offset, but does contain the
    #  interactions, which we need.
    factors <- attr(Terms, 'factors')
    if (any(factors[tcl,] >1)) stop("cluster() cannot be in an interaction")
    if (attr(Terms, "response") ==0)
        stop("formula must have a Surv response")
    # reformulate with the response option puts ` ` around Surv, which messes
    #  up evaluation, hence the fancy dance to replace a piece rather
    #  than recreate
    temp <- attr(Terms, "term.labels")
    oo <- attr(Terms, 'specials')$offset
    if (!is.null(oo)) {
        # add the offset to the set of labels
        ooterm <- rownames(factors)[oo]
        if (oo < tcl) temp <- c(ooterm, temp)
        else temp <- c(temp, ooterm)
    }
    if (is.null(Call$cluster))
        Call$cluster <- attr(Terms, "variables")[[1+tcl]][[2]]
    else warning("cluster appears both in a formula and as an argument, formula term ignored")
    if (is.list(formula))
         formula[[1]][[3]] <- reformulate(temp[1-tcl])[[2]]
    else formula[[3]]      <- reformulate(temp[1-tcl])[[2]]
```

```
    Call$formula <- formula

}

# create a call to model.frame() that contains the formula (required)
#  and any other of the relevant optional arguments
#  but don't evaluate it just yet
indx <- match(c("formula", "data", "weights", "subset", "na.action",
                "cluster", "id", "istate"),
              names(Call), nomatch=0)
if (indx[1] ==0) stop("A formula argument is required")
tform <- Call[c(1,indx)]  # only keep the arguments we wanted
tform[[1L]] <- quote(stats::model.frame)  # change the function called

# if the formula is a list, do the first level of processing on it.
if (is.list(formula)) {
    ⟨coxph-multiform1⟩
}
else {
    multiform <- FALSE   # formula is not a list of expressions
    covlist <- NULL
    dformula <- formula
}

# add specials to the formula
special <- c("strata", "tt", "frailty", "ridge", "pspline")
tform$formula <- if(missing(data)) terms(formula, special) else
                                   terms(formula, special, data=data)

# Make "tt" visible for coxph formulas, without making it visible elsewhere
if (!is.null(attr(tform$formula, "specials")$tt)) {
    coxenv <- new.env(parent= environment(formula))
    assign("tt", function(x) x, envir=coxenv)
    environment(tform$formula) <- coxenv
}

# okay, now evaluate the formula
mf <- eval(tform, parent.frame())
Terms <- terms(mf)
attr(Terms,'term.labels') = gsub('\n',' ', attr(Terms,'term.labels'))
colnames(attr(Terms,'factors')) = gsub('\n',' ', colnames(attr(Terms,'factors')))
rownames(attr(Terms,'factors')) = gsub('\n',' ', rownames(attr(Terms,'factors')))

# Grab the response variable, and deal with Surv2 objects
n <- nrow(mf)
```

```
Y <- model.response(mf)
isSurv2 <- inherits(Y, "Surv2")
if (isSurv2) {
    # this is Surv2 style data
    # if there were any obs removed due to missing, remake the model frame
    if (length(attr(mf, "na.action"))) {
        tform$na.action <- na.pass
        mf <- eval.parent(tform)
    }
    if (!is.null(attr(Terms, "specials")$cluster))
        stop("cluster() cannot appear in the model statement")
    new <- surv2data(mf)
    mf <- new$mf
    istate <- new$istate
    id <- new$id
    Y <- new$y
    n <- nrow(mf)
}
else {
    if (!is.Surv(Y)) stop("Response must be a survival object")
    id <- model.extract(mf, "id")
    istate <- model.extract(mf, "istate")
}
if (n==0) stop("No (non-missing) observations")

type <- attr(Y, "type")
multi <- FALSE
if (type=="mright" || type == "mcounting") multi <- TRUE
else if (type!='right' && type!='counting')
    stop(paste("Cox model doesn't support \"", type,
                       "\" survival data", sep=''))
data.n <- nrow(Y)   #remember this before any time transforms

if (!multi && multiform)
    stop("formula is a list but the response is not multi-state")
if (multi && length(attr(Terms, "specials")$frailty) >0)
    stop("multi-state models do not currently support frailty terms")
if (multi && length(attr(Terms, "specials")$pspline) >0)
    stop("multi-state models do not currently support pspline terms")
if (multi && length(attr(Terms, "specials")$ridge) >0)
    stop("multi-state models do not currently support ridge penalties")

if (control$timefix) Y <- aeqSurv(Y)
⟨coxph-bothsides⟩

# The time transform will expand the data frame mf.  To do this
```

```
#  it needs Y and the strata.  Everything else (cluster, offset, weights)
#  should be extracted after the transform
#
strats <- attr(Terms, "specials")$strata
hasinteractions <- FALSE
dropterms <- NULL
if (length(strats)) {
    stemp <- untangle.specials(Terms, 'strata', 1)
    if (length(stemp$vars)==1) strata.keep <- mf[[stemp$vars]]
    else strata.keep <- strata(mf[,stemp$vars], shortlabel=TRUE)
    istrat <- as.integer(strata.keep)

    for (i in stemp$vars) {  #multiple strata terms are allowed
        # The factors attr has one row for each variable in the frame, one
        #   col for each term in the model.  Pick rows for each strata
        #   var, and find if it participates in any interactions.
        if (any(attr(Terms, 'order')[attr(Terms, "factors")[i,] >0] >1))
            hasinteractions <- TRUE
    }
    if (!hasinteractions) dropterms <- stemp$terms
} else istrat <- NULL

if (hasinteractions && multi)
    stop("multi-state coxph does not support strata*covariate interactions")


timetrans <- attr(Terms, "specials")$tt
if (missing(tt)) tt <- NULL
if (length(timetrans)) {
    if (multi || isSurv2) stop("the tt() transform is not implemented for multi-state or Surv
     ⟨coxph-transform⟩
    }

xlevels <- .getXlevels(Terms, mf)

# grab the cluster, if present.  Using cluster() in a formula is no
#  longer encouraged
cluster <- model.extract(mf, "cluster")
weights <- model.weights(mf)
# The user can call with cluster, id, robust, or any combination
# Default for robust: if cluster or any id with > 1 event or
#  any weights that are not 0 or 1, then TRUE
# If only id, treat it as the cluster too
has.cluster <- !(missing(cluster) || length(cluster)==0)
has.id <-      !(missing(id) || length(id)==0)
has.rwt<-      (!is.null(weights) && any(weights != floor(weights)))
```

```
#has.rwt<- FALSE  # we are rethinking this
has.robust <-  (!missing(robust) && !is.null(robust))  # arg present
if (has.id) id <- as.factor(id)

if (missing(robust) || is.null(robust)) {
    if (has.cluster || has.rwt ||
             (has.id && (multi || anyDuplicated(id[Y[,ncol(Y)]==1]))))
        robust <- TRUE else robust <- FALSE
}
if (!is.logical(robust)) stop("robust must be TRUE/FALSE")

if (has.cluster) {
    if (!robust) {
        warning("cluster specified with robust=FALSE, cluster ignored")
        ncluster <- 0
        clname <- NULL
    }
    else {
        if (is.factor(cluster)) {
            clname <- levels(cluster)
            cluster <- as.integer(cluster)
        } else {
            clname  <- sort(unique(cluster))
            cluster <- match(cluster, clname)
        }
        ncluster <- length(clname)
    }
} else {
    if (robust && has.id) {
        # treat the id as both identifier and clustering
        clname <- levels(id)
        cluster <- as.integer(id)
        ncluster <- length(clname)
    }
    else {
        ncluster <- 0  # has neither
    }
}

# if the user said "robust", (time1,time2) data, and no cluster or
#  id, complain about it
if (robust && is.null(cluster)) {
    if (ncol(Y) ==2 || !has.robust) cluster <- seq.int(1, nrow(mf))
    else stop("one of cluster or id is needed")
}
```

```
contrast.arg <- NULL  #due to shared code with model.matrix.coxph
attr(Terms, "intercept") <- 1  # always have a baseline hazard

if (multi) {
    ⟨coxph-multiform2⟩
}

⟨coxph-make-X⟩
⟨coxph-setup⟩
if (multi) {
    ⟨coxph-multi-X⟩
}

# infinite covariates are not screened out by the na.omit routines
#  But this needs to be done after the multi-X part
if (!all(is.finite(X)))
    stop("data contains an infinite predictor")


# init is checked after the final X matrix has been made
if (missing(init)) init <- NULL
else {
    if (length(init) != ncol(X)) stop("wrong length for init argument")
    temp <- X %*% init - sum(colMeans(X) * init) + offset
    # it's okay to have a few underflows, but if all of them are too
    #   small we get all zeros
    if (any(exp(temp) > .Machine$double.xmax) || all(exp(temp)==0))
    stop("initial values lead to overflow or underflow of the exp function")
}

⟨coxph-penal⟩
⟨coxph-compute⟩
⟨coxph-finish⟩
}
```

Multi-state models have a multi-state response, optionally they have a formula that is a list. If the formula is a list then the first element is the default formula with a survival response and covariates on the right. Further elements are of the form from/to covariates / options and specify other covariates for all from:to transitions. Steps in processing such a formula are

1. Gather all the variables that appear on a right-hand side, and create a master formula y all of them. This is used to create the model.frame. We also need to defer missing value processing, since some covariates might appear for only some transitions.
2. Get the data. The response, id, and statedata variables can now be checked for consistency with the formulas.
3. After X has been formed, expand it.

Here is code for the first step.

⟨*coxph-multiform1*⟩=
```
 multiform <- TRUE
 dformula <- formula[[1]]   # the default formula for transitions
 if (missing(statedata)) covlist <- parsecovar1(formula[-1])
 else {
     if (!inherits(statedata, "data.frame"))
         stop("statedata must be a data frame")
     if (is.null(statedata$state))
         stop("statedata data frame must contain a 'state' variable")
     covlist <- parsecovar1(formula[-1], names(statedata))
 }

 # create the master formula, used for model.frame
 # the term.labels + reformulate + environment trio is used in [.terms;
 #  if it's good enough for base R it's good enough for me
 tlab <- unlist(lapply(covlist$rhs, function(x)
     attr(terms.formula(x$formula), "term.labels")))
 tlab <- c(attr(terms.formula(dformula), "term.labels"), tlab)
 newform <- reformulate(tlab, dformula[[2]])
 environment(newform) <- environment(dformula)
 formula <- newform
 tform$na.action <- na.pass  # defer any missing value work to later
```

⟨*coxph-multiform2*⟩=
```
 # check for consistency of the states, and create a transition
 #  matrix
 if (length(id)==0)
     stop("an id statement is required for multi-state models")

 mcheck <- survcheck2(Y, id, istate)
 # error messages here
 if (mcheck$flag["overlap"] > 0)
     stop("data set has overlapping intervals for one or more subjects")

 transitions <- mcheck$transitions
 istate <- mcheck$istate
 states <- mcheck$states

 #  build tmap, which has one row per term, one column per transition
 if (missing(statedata))
     covlist2 <- parsecovar2(covlist, NULL, dformula= dformula,
                          Terms, transitions, states)
 else covlist2 <- parsecovar2(covlist, statedata, dformula= dformula,
                          Terms, transitions, states)
```

```
tmap <- covlist2$tmap
if (!is.null(covlist)) {
    ⟨coxph-missing⟩
}
```

For multi-state models we can't tell what observations should be removed until any extra formulas have been processed. There may be rows that are missing *some* of the covariates but are okay for *some* transitions. Others could be useless. Those rows can be removed from the model frame before creating the X matrix. Also identify partially used rows, ones where the necessary covariates are present for some of the possible transitions but not all. Those obs are dealt with later by the stacker function.

⟨*coxph-missing*⟩=

```
# first vector will be true if there is at least 1 transition for which all
#  covariates are present, second if there is at least 1 for which some are not
good.tran <- bad.tran <- rep(FALSE, nrow(Y))
# We don't need to check interaction terms
termname <- rownames(attr(Terms, 'factors'))
trow <- (!is.na(match(rownames(tmap), termname)))

# create a missing indicator for each term
termiss <- matrix(0L, nrow(mf), ncol(mf))
for (i in 1:ncol(mf)) {
    xx <- is.na(mf[[i]])
    if (is.matrix(xx)) termiss[,i] <- apply(xx, 1, any)
    else termiss[,i] <- xx
}

for (i in levels(istate)) {
    rindex <- which(istate ==i)
    j <- which(covlist2$mapid[,1] == match(i, states))  #possible transitions
    for (jcol in j) {
        k <- which(trow & tmap[,jcol] > 0)  # the terms involved in that
        bad.tran[rindex] <- (bad.tran[rindex] |
                             apply(termiss[rindex, k, drop=FALSE], 1, any))
        good.tran[rindex] <- (good.tran[rindex] |
                              apply(!termiss[rindex, k, drop=FALSE], 1, all))
    }
}
n.partially.used <- sum(good.tran & bad.tran & !is.na(Y))
omit <- (!good.tran & bad.tran) | is.na(Y)
if (all(omit)) stop("all observations deleted due to missing values")
temp <- setNames(seq(omit)[omit], attr(mf, "row.names")[omit])
attr(temp, "class") <- "omit"
mf <- mf[!omit,, drop=FALSE]
attr(mf, "na.action") <- temp
Y <- Y[!omit]
```

```
id <- id[!omit]
if (length(istate)) istate <- istate[!omit]  # istate can be NULL
```

For a multi-state model, create the expanded X matrix. Sometimes it is much expanded. The first step is to create the cmap matrix from tmap by expanding terms; factors turn into multiple columns for instance. If tmap has rows (terms) for strata, then we have to deal with the complication that a strata might be applied to some transitions and not to others.

⟨*coxph-multi-X*⟩=
```
if (length(strats) >0) {
    stratum_map <- tmap[c(1L, strats),] # strats includes Y, + tmap has an extra row
    stratum_map[-1,] <- ifelse(stratum_map[-1,] >0, 1L, 0L)
    if (nrow(stratum_map) > 2) {
        temp <- stratum_map[-1,]
        if (!all(apply(temp, 2, function(x) all(x==0) || all(x==1)))) {
            # the hard case: some transitions use one strata variable, some
            #  transitions use another.  We need to keep them separate
            strata.keep <- mf[,strats]  # this will be a data frame
            istrat <- sapply(strata.keep, as.numeric)
        }
    }
}
else stratum_map <- tmap[1,,drop=FALSE]
```

Also create the initial values vector.

The stacker function will create a separate block of observations for every unique value in `stratum_map`. Now say that two transitions A:B and A:C share the same baseline hazard. Then either a B or a C outcome will be an "event" in that stratum; they would only be distinguished by perhaps having different covariates. The first thing we do with the result is to rebuild the transitions matrix: the working version was created before removing missings and can seriously overstate the number of transitions available. Then set up the data.

⟨*coxph-multi-X*⟩=
```
cmap <- parsecovar3(tmap, colnames(X), attr(X, "assign"), covlist2$phbaseline)
xstack <- stacker(cmap, stratum_map, as.integer(istate), X, Y, strata=istrat,
                  states=states)

rkeep <- unique(xstack$rindex)
transitions <- survcheck2(Y[rkeep,], id[rkeep], istate[rkeep])$transitions

X <- xstack$X
Y <- xstack$Y
istrat <- xstack$strata
if (length(offset)) offset <- offset[xstack$rindex]
if (length(weights)) weights <- weights[xstack$rindex]
if (length(cluster)) cluster <- cluster[xstack$rindex]
```

The next step for multi X is to remake the assign attribute. It is a list with one element per term, and needs to be expanded in the same way as `tmap`, which has one row per term (+ an intercept row). For `predict, type='terms'` to work, no label can be repeated in the final assign object. If a variable 'fred' were common across all the states we would want to use that as the label, but if it appears twice, as separate terms for two different transitions, then we label it as fred_x:y where x:y is the transition.

⟨*coxph-multi-X*⟩=
```
t2 <- tmap[-c(1, strats),,drop=FALSE]   # remove the intercept row and strata rows
r2 <- row(t2)[!duplicated(as.vector(t2)) & t2 !=0]
c2 <- col(t2)[!duplicated(as.vector(t2)) & t2 !=0]
a2 <- lapply(seq(along.with=r2), function(i) {cmap[assign[[r2[i]]], c2[i]]})
# which elements are unique?
tab <- table(r2)
count <- tab[r2]
names(a2) <- ifelse(count==1, row.names(t2)[r2],
                    paste(row.names(t2)[r2], colnames(cmap)[c2], sep="_"))
assign <- a2
```

An increasingly common error is for users to put the time variable on both sides of the formula, in the mistaken idea that this will deal with a failure of proportional hazards. Add a test for such models, but don't bail out. There will be cases where someone has the the stop variable in an expression on the right hand side, to create current age say. The `variables` attribute of the Terms object is the expression form of a list that contains the response variable followed by the predictors. Subscripting this, element 1 is the call to "list" itself so we always retain it. My `terms.inner` function works only with formula objects.

⟨*coxph-bothsides*⟩=
```
if (length(attr(Terms, 'variables')) > 2) { # a ~1 formula has length 2
    ytemp <- terms.inner(formula[1:2])
    suppressWarnings(z <- as.numeric(ytemp)) # are any of the elements numeric?
    ytemp <- ytemp[is.na(z)]  # toss numerics, e.g. Surv(t, 1-s)
    xtemp <- terms.inner(formula[-2])
    if (any(!is.na(match(xtemp, ytemp))))
        warning("a variable appears on both the left and right sides of the formula")
}
```

At this point we deal with any time transforms. The model frame is expanded to a "fake" data set that has a separate stratum for each unique event-time/strata combination, and any tt() terms in the formula are processed. The first step is to create the index vector `tindex` and new strata `.strata.`. This last is included in a model.frame call (for others to use), internally the code simply replaces the `istrat` variable. A (modestly) fast C-routine first counts up and indexes the observations. We start out with error checks; since the computation can be slow we want to complain early.

⟨*coxph-transform*⟩=
```
timetrans <- untangle.specials(Terms, 'tt')
ntrans <- length(timetrans$terms)
```

```
if (is.null(tt)) {
    tt <- function(x, time, riskset, weights){ #default to O'Brien's logit rank
        obrien <- function(x) {
            r <- rank(x)
            (r-.5)/(.5+length(r)-r)
        }
        unlist(tapply(x, riskset, obrien))
    }
}
if (is.function(tt)) tt <- list(tt)  #single function becomes a list

if (is.list(tt)) {
    if (any(!sapply(tt, is.function)))
        stop("The tt argument must contain function or list of functions")
    if (length(tt) != ntrans) {
        if (length(tt) ==1) {
            temp <- vector("list", ntrans)
            for (i in 1:ntrans) temp[[i]] <- tt[[1]]
            tt <- temp
        }
        else stop("Wrong length for tt argument")
    }
}
else stop("The tt argument must contain a function or list of functions")

if (ncol(Y)==2) {
    if (length(strats)==0) {
        sorted <- order(-Y[,1], Y[,2])
        newstrat <- rep.int(0L, nrow(Y))
        newstrat[1] <- 1L
        }
    else {
        sorted <- order(istrat, -Y[,1], Y[,2])
        #newstrat marks the first obs of each strata
        newstrat <-  as.integer(c(1, 1*(diff(istrat[sorted])!=0)))
        }
    if (storage.mode(Y) != "double") storage.mode(Y) <- "double"
    counts <- .Call(Ccoxcount1, Y[sorted,],
                    as.integer(newstrat))
    tindex <- sorted[counts$index]
}
else {
    if (length(strats)==0) {
        sort.end  <- order(-Y[,2], Y[,3])
        sort.start<- order(-Y[,1])
```

```
        newstrat  <- c(1L, rep(0, nrow(Y) -1))
    }
    else {
        sort.end  <- order(istrat, -Y[,2], Y[,3])
        sort.start<- order(istrat, -Y[,1])
        newstrat  <- c(1L, as.integer(diff(istrat[sort.end])!=0))
    }
    if (storage.mode(Y) != "double") storage.mode(Y) <- "double"
    counts <- .Call(Ccoxcount2, Y,
                    as.integer(sort.start -1L),
                    as.integer(sort.end -1L),
                    as.integer(newstrat))
    tindex <- counts$index
}
```

The C routine has returned a list with 4 elements

nrisk a vector containing the number at risk at each event time

time the vector of event times

status a vector of status values

index a vector containing the set of subjects at risk for event time 1, followed by those at risk at event time 2, those at risk at event time 3, etc.

The new data frame is then a simple creation. The subtle part below is a desire to retain transformation information so that a downstream call to `termplot` will work. The tt function supplied by the user often finishes with a call to `pspline` or `ns`. If the returned value of the `tt` call has a class for which a `makepredictcall` method exists then we need to do 2 things:

1. Construct a fake call, e.g., "pspline(age)", then feed it and the result of tt as arguments to `makepredictcall`
2. Replace that componenent in the predvars attribute of the terms.

The `timetrans$terms` value is a count of the right hand side of the formula. Some objects in the terms structure are unevaluated calls that include y, this adds 2 to the count (the call to "list" and the response).

⟨*coxph-transform*⟩=

```
Y <- Surv(rep(counts$time, counts$nrisk), counts$status)
type <- 'right'  # new Y is right censored, even if the old was (start, stop]

mf <- mf[tindex,]
istrat <- rep(1:length(counts$nrisk), counts$nrisk)
weights <- model.weights(mf)
if (!is.null(weights) && any(!is.finite(weights)))
    stop("weights must be finite")
```

```
tcall <- attr(Terms, 'variables')[timetrans$terms+2]
pvars <- attr(Terms, 'predvars')
pmethod <- sub("makepredictcall.", "", as.vector(methods("makepredictcall")))
for (i in 1:ntrans) {
    newtt <- (tt[[i]])(mf[[timetrans$var[i]]], Y[,1], istrat, weights)
    mf[[timetrans$var[i]]] <- newtt
    nclass <- class(newtt)
    if (any(nclass %in% pmethod)) { # It has a makepredictcall method
        dummy <- as.call(list(as.name(class(newtt)[1]), tcall[[i]][[2]]))
        ptemp <- makepredictcall(newtt, dummy)
        pvars[[timetrans$terms[i]+2]] <- ptemp
    }
}
attr(Terms, "predvars") <- pvars
```

This is the C code for time-transformation. For the first case it expects y to contain time and status sorted from longest time to shortest, and strata=1 for the first observation of each strata.

⟨*coxcount1*⟩=

```
#include "survS.h"
/*
** Count up risk sets and identify who is in each
*/
SEXP coxcount1(SEXP y2, SEXP strat2) {
    int ntime, nrow;
    int i, j, n;
    int stratastart=0;  /* start row for this strata */
    int nrisk=0;  /* number at risk (=0 to stop -Wall complaint)*/
    double *time, *status;
    int *strata;
    double dtime;
    SEXP rlist, rlistnames, rtime, rn, rindex, rstatus;
    int *rrindex, *rrstatus;

    n = nrows(y2);
    time = REAL(y2);
    status = time +n;
    strata = INTEGER(strat2);

    /*
    ** First pass: count the total number of death times (risk sets)
    **  and the total number of rows in the new data set.
    */
    ntime=0; nrow=0;
    for (i=0; i<n; i++) {
```

```
        if (strata[i] ==1) nrisk =0;
        nrisk++;
        if (status[i] ==1) {
            ntime++;
            dtime = time[i];
            /* walk across tied times, if any */
            for (j=i+1; j<n && time[j]==dtime && status[j]==1 && strata[j]==0;
                 j++) nrisk++;
            i = j-1;
            nrow += nrisk;
        }
    }
    ⟨coxcount-alloc-memory⟩

    /*
    ** Pass 2, fill them in
    */
    ntime=0;
    for (i=0; i<n; i++) {
        if (strata[i] ==1) stratastart =i;
        if (status[i]==1) {
            dtime = time[i];
            for (j=stratastart; j<i; j++) *rrstatus++=0; /*non-deaths */
            *rrstatus++ =1; /* this death */
            /* tied deaths */
            for(j= i+1; j<n && status[j]==1 && time[j]==dtime  && strata[j]==0;
                j++) *rrstatus++ =1;
            i = j-1;

            REAL(rtime)[ntime] = dtime;
            INTEGER(rn)[ntime] = i +1 -stratastart;
            ntime++;
            for (j=stratastart; j<=i; j++) *rrindex++ = j+1;
            }
    }
    ⟨coxcount-list-return⟩
}
```

The start-stop case is a bit more work. The set of subjects still at risk is an arbitrary set so we have to keep an index vector `atrisk`. At each new death time we write out the set of those at risk, with the deaths last. I toyed with the idea of a binary tree then realized it was not useful: at each death we need to list out all the subjects at risk into the index vector which is an $O(n)$ process, tree or not.

⟨*coxcount1*⟩=
```
#include "survS.h"
/* count up risk sets and identify who is in each, (start,stop] version */
```

```
SEXP coxcount2(SEXP y2, SEXP isort1, SEXP isort2, SEXP strat2) {
    int ntime, nrow;
    int i, j, istart, n;
    int nrisk=0, *atrisk;
    double *time1, *time2, *status;
    int *strata;
    double dtime;
    int iptr, jptr;

    SEXP rlist, rlistnames, rtime, rn, rindex, rstatus;
    int *rrindex, *rrstatus;
    int *sort1, *sort2;

    n = nrows(y2);
    time1 = REAL(y2);
    time2 =  time1+n;
    status = time2 +n;
    strata = INTEGER(strat2);
    sort1 = INTEGER(isort1);
    sort2 = INTEGER(isort2);

    /*
    ** First pass: count the total number of death times (risk sets)
    **  and the total number of rows in the new data set
    */
    ntime=0; nrow=0;
    istart =0;  /* walks along the sort1 vector (start times) */
        for (i=0; i<n; i++) {
        iptr = sort2[i];
        if (strata[i]==1) nrisk=0;
        nrisk++;
        if (status[iptr] ==1) {
            ntime++;
            dtime = time2[iptr];
            for (; istart <i && time1[sort1[istart]] >= dtime; istart++)
                        nrisk--;
            for(j= i+1; j<n; j++) {
                jptr = sort2[j];
                if (status[jptr]==1 && time2[jptr]==dtime && strata[jptr]==0)
                    nrisk++;
                else break;
                }
            i= j-1;
            nrow += nrisk;
            }
        }
```

```
⟨coxcount-alloc-memory⟩
atrisk = (int *)R_alloc(n, sizeof(int)); /* marks who is at risk */

/*
** Pass 2, fill them in
*/
ntime=0; nrisk=0;
j=0;  /* pointer to time1 */;
istart=0;
for (i=0; i<n; ) {
    iptr = sort2[i];
    if (strata[i] ==1) {
        nrisk=0;
        for (j=0; j<n; j++) atrisk[j] =0;
        }
    nrisk++;
    if (status[iptr]==1) {
        dtime = time2[iptr];
        for (; istart<i && time1[sort1[istart]] >=dtime; istart++) {
            atrisk[sort1[istart]]=0;
            nrisk--;
            }
        for (j=1; j<nrisk; j++) *rrstatus++ =0;
        for (j=0; j<n; j++) if (atrisk[j]) *rrindex++ = j+1;

        atrisk[iptr] =1;
        *rrstatus++ =1;
        *rrindex++ = iptr +1;
        for (j=i+1; j<n; j++) {
            jptr = sort2[j];
            if (time2[jptr]==dtime && status[jptr]==1 && strata[jptr]==0){
                atrisk[jptr] =1;
                *rrstatus++ =1;
                *rrindex++ = jptr +1;
                nrisk++;
                }
            else break;
            }
        i = j;
        REAL(rtime)[ntime] = dtime;
        INTEGER(rn)[ntime] = nrisk;
        ntime++;
    }
    else {
        atrisk[iptr] =1;
```

```
            i++;
        }
    }
    ⟨coxcount-list-return⟩
}
```

⟨*coxcount-alloc-memory*⟩=

```
/*
**  Allocate memory
*/
PROTECT(rtime = allocVector(REALSXP, ntime));
PROTECT(rn = allocVector(INTSXP, ntime));
PROTECT(rindex=allocVector(INTSXP, nrow));
PROTECT(rstatus=allocVector(INTSXP,nrow));
rrindex = INTEGER(rindex);
rrstatus= INTEGER(rstatus);
```

⟨*coxcount-list-return*⟩=

```
/* return the list */
PROTECT(rlist = allocVector(VECSXP, 4));
SET_VECTOR_ELT(rlist, 0, rn);
SET_VECTOR_ELT(rlist, 1, rtime);
SET_VECTOR_ELT(rlist, 2, rindex);
SET_VECTOR_ELT(rlist, 3, rstatus);
PROTECT(rlistnames = allocVector(STRSXP, 4));
SET_STRING_ELT(rlistnames, 0, mkChar("nrisk"));
SET_STRING_ELT(rlistnames, 1, mkChar("time"));
SET_STRING_ELT(rlistnames, 2, mkChar("index"));
SET_STRING_ELT(rlistnames, 3, mkChar("status"));
setAttrib(rlist, R_NamesSymbol, rlistnames);

unprotect(6);
return(rlist);
```

We now return to the original thread of the program, though perhaps with new data, and build the X matrix. Creation of the X matrix for a Cox model requires just a bit of trickery. The baseline hazard for a Cox model plays the role of an intercept, but does not appear in the X matrix. However, to create the columns of X for factor variables correctly, we need to call the model.matrix routine in such a way that it *thinks* there is an intercept, and so we set the intercept attribute to 1 in the terms object before calling model.matrix, ignoring any -1 term the user may have added. One simple way to handle all this is to call model.matrix on the original formula and then remove the terms we don't need. However,

1. The cluster() term, if any, could lead to thousands of extraneous "intercept" columns which are never needed.
2. Likewise, nested case-control models can have thousands of strata, again leading many intercepts we never need. They never have strata by covariate interactions, however.

3. If there are strata by covariate interactions in the model, the dummy intercepts-per-strata columns are necessary information for the model.matrix routine to correctly compute other columns of X.

On later reflection `cluster` should never have been in the model statement in the first place, something that became painfully apparent with addition of multi-state models. In the future we will discourage it. For reason 2 above the usual plan is to also remove strata terms from the "Terms" object *before* calling model.matrix, unless there are strata by covariate interactions in which case we remove them after. If anything is pre-dropped, for documentation purposes we want the returned assign attribute to match the Terms structure that we will hand back. (Do we ever use it?) In particular, the numbers therein correspond to the column names in `attr(Terms, 'factors')` The requires a shift. The cluster and strata terms are seen as main effects, so appear early in that list. We have found a case where terms get relabeled:

⟨*relabel*⟩=
```
t1 <- terms( ~(x1 + x2):x3 + strata(x4))
t2 <- terms( ~(x1 + x2):x3)
t3 <- t1[-1]
colnames(attr(t1, "factors"))
colnames(attr(t2, "factors"))
colnames(attr(t3, "factors"))
```

In t1 the strata term appears first, as it is the only thing that looks like a main effect, and the column labels are strata(x4), x1:x3, x2:x3. In t3 the column labels are x1:x3 and x3:x2 — note left-right swap of the second. This means that using match() on the labels is not a reliable approach. We instead assume that nothing is reordered and do a shift.

⟨*coxph-make-X*⟩=
```
if (length(dropterms)) {
    Terms2 <- Terms[ -dropterms]
    X <- model.matrix(Terms2, mf, constrasts.arg=contrast.arg)
    # we want to number the terms wrt the original model matrix
    temp <- attr(X, "assign")
    shift <- sort(dropterms)
    for (i in seq(along.with=shift))
        temp <- temp + 1*(shift[i] <= temp)
    attr(X, "assign") <- temp
}
else X <- model.matrix(Terms, mf, contrasts.arg=contrast.arg)

# drop the intercept after the fact, and also drop strata if necessary
Xatt <- attributes(X)
if (hasinteractions) adrop <- c(0, untangle.specials(Terms, "strata")$terms)
else adrop <- 0
xdrop <- Xatt$assign %in% adrop  #columns to drop (always the intercept)
X <- X[, !xdrop, drop=FALSE]
```

```
 attr(X, "assign") <- Xatt$assign[!xdrop]
 attr(X, "contrasts") <- Xatt$contrasts
```

Finish the setup. If someone includes an init statement or offset, make sure that it does not lead to instant code failure due to overflow/underflow.

⟨*coxph-setup*⟩=
```
 offset <- model.offset(mf)
 if (is.null(offset) | all(offset==0)) offset <- rep(0., nrow(mf))
 else if (any(!is.finite(offset) | !is.finite(exp(offset))))
     stop("offsets must lead to a finite risk score")

 weights <- model.weights(mf)
 if (!is.null(weights) && any(!is.finite(weights)))
     stop("weights must be finite")

 assign <- attrassign(X, Terms)
 contr.save <- attr(X, "contrasts")
 ⟨coxph-zeroevent⟩
```

Check for a rare edge case: a data set with no events. In this case the return structure is simple. The coefficients will all be NA, since they can't be estimated. The variance matrix is all zeros, in line with the usual rule to zero out any row and col corresponding to an NA coef. The loglik is the sum of zero terms, which we set to zero like the usual R result for sum(numeric(0)). An overall idea is to return something that won't blow up later code.

⟨*coxph-zeroevent*⟩=
```
 if (sum(Y[, ncol(Y)]) == 0) {
     # No events in the data!
     ncoef <- ncol(X)
     ctemp <- rep(NA, ncoef)
     names(ctemp) <- colnames(X)
     concordance= c(concordant=0, discordant=0, tied.x=0, tied.y=0, tied.xy=0,
                    concordance=NA, std=NA, timefix=FALSE)
     rval <- list(coefficients= ctemp,
                  var = matrix(0.0, ncoef, ncoef),
                  loglik=c(0,0),
                  score =0,
                  iter =0,
                  linear.predictors = offset,
                  residuals = rep(0.0, data.n),
                  means = colMeans(X), method=method,
                  n = data.n, nevent=0, terms=Terms, assign=assign,
                  concordance=concordance,  wald.test=0.0,
                  y = Y, call=Call)
     class(rval) <- "coxph"
     return(rval)
 }
```

Check for penalized terms in the model, and set up infrastructure for the fitting routines to deal with them.

⟨*coxph-penal*⟩=
```
 pterms <- sapply(mf, inherits, 'coxph.penalty')
 if (any(pterms)) {
     pattr <- lapply(mf[pterms], attributes)
     pname <- names(pterms)[pterms]
     #
     # Check the order of any penalty terms
     ord <- attr(Terms, "order")[match(pname, attr(Terms, 'term.labels'))]
     if (any(ord>1)) stop ('Penalty terms cannot be in an interaction')
     pcols <- assign[match(pname, names(assign))]

     fit <- coxpenal.fit(X, Y, istrat, offset, init=init,
                         control,
                         weights=weights, method=method,
                         row.names(mf), pcols, pattr, assign,
                         nocenter= nocenter)
 }
```

⟨*coxph-compute*⟩=
```
 else {
     rname <- row.names(mf)
     if (multi) rname <- rname[xstack$rindex]
     if( method=="breslow" || method =="efron") {
         if (grepl('right', type))
             fit <- coxph.fit(X, Y, istrat, offset, init, control,
                              weights=weights, method=method,
                              rname, nocenter=nocenter)
         else  fit <- agreg.fit(X, Y, istrat, offset, init, control,
                                weights=weights, method=method,
                                rname, nocenter=nocenter)
     }
     else if (method=='exact') {
         if (type== "right")
             fit <- coxexact.fit(X, Y, istrat, offset, init, control,
                                 weights=weights, method=method,
                                 rname, nocenter=nocenter)
         else fit <- agexact.fit(X, Y, istrat, offset, init, control,
                                 weights=weights, method=method,
                                 rname, nocenter=nocenter)
     }
     else stop(paste ("Unknown method", method))
 }
```

⟨*coxph-finish*⟩=

```
if (is.character(fit)) {
    fit <- list(fail=fit)
    class(fit) <- 'coxph'
}
else {
    if (!is.null(fit$coefficients) && any(is.na(fit$coefficients))) {
       vars <- (1:length(fit$coefficients))[is.na(fit$coefficients)]
       msg <-paste("X matrix deemed to be singular; variable",
                       paste(vars, collapse=" "))
       if (!singular.ok) stop(msg)
       # else warning(msg)  # stop being chatty
    }
    fit$n <- data.n
    fit$nevent <- sum(Y[,ncol(Y)])
    fit$terms <- Terms
    fit$assign <- assign
    class(fit) <- fit$class
    fit$class <- NULL

    # don't compute a robust variance if there are no coefficients
    if (robust && !is.null(fit$coefficients) && !all(is.na(fit$coefficients))) {
        fit$naive.var <- fit$var
        # a little sneaky here: by calling resid before adding the
        #   na.action method, I avoid having missings re-inserted
        # I also make sure that it doesn't have to reconstruct X and Y
        fit2 <- c(fit, list(x=X, y=Y, weights=weights))
        if (length(istrat)) fit2$strata <- istrat
        if (length(cluster)) {
            temp <- residuals.coxph(fit2, type='dfbeta', collapse=cluster,
                                    weighted=TRUE)
            # get score for null model
            if (is.null(init))
                    fit2$linear.predictors <- 0*fit$linear.predictors
            else fit2$linear.predictors <- c(X %*% init)
            temp0 <- residuals.coxph(fit2, type='score', collapse=cluster,
                                     weighted=TRUE)
        }
        else {
            temp <- residuals.coxph(fit2, type='dfbeta', weighted=TRUE)
            fit2$linear.predictors <- 0*fit$linear.predictors
            temp0 <- residuals.coxph(fit2, type='score', weighted=TRUE)
        }
        fit$var <- t(temp) %*% temp
        u <- apply(as.matrix(temp0), 2, sum)
        fit$rscore <- coxph.wtest(t(temp0)%*%temp0, u, control$toler.chol)$test
    }
```

```
    #Wald test
    if (length(fit$coefficients) && is.null(fit$wald.test)) {
        #not for intercept only models, or if test is already done
        nabeta <- !is.na(fit$coefficients)
        # The init vector might be longer than the betas, for a sparse term
        if (is.null(init)) temp <- fit$coefficients[nabeta]
        else temp <- (fit$coefficients -
                      init[1:length(fit$coefficients)])[nabeta]
        fit$wald.test <-  coxph.wtest(fit$var[nabeta,nabeta], temp,
                                      control$toler.chol)$test
    }

    # Concordance.  Done here so that we can use cluster if it is present
    # The returned value is a subset of the full result, partly because it
    #  is all we need, but more for backward compatability with survConcordance.fit
    if (length(cluster))
        temp <- concordancefit(Y, fit$linear.predictors, istrat, weights,
                                          cluster=cluster, reverse=TRUE,
                               timefix= FALSE)
    else temp <- concordancefit(Y, fit$linear.predictors, istrat, weights,
                                reverse=TRUE, timefix= FALSE)
    if (is.matrix(temp$count))
         fit$concordance <- c(colSums(temp$count), concordance=temp$concordance,
                              std=sqrt(temp$var))
    else fit$concordance <- c(temp$count, concordance=temp$concordance,
                              std=sqrt(temp$var))

    na.action <- attr(mf, "na.action")
    if (length(na.action)) fit$na.action <- na.action
    if (model) {
        if (length(timetrans)) {
            stop("'model=TRUE' not supported for models with tt terms")
        }
        fit$model <- mf
    }
    if (x)  {
        fit$x <- X
        if (length(timetrans)) fit$strata <- istrat
        else if (length(strats)) fit$strata <- strata.keep
    }
    if (y)  fit$y <- Y
    fit$timefix <- control$timefix  # remember this option
}
```

If any of the weights were not 1, save the results. Add names to the means component, which

are occassionally useful to survfit.coxph. Other objects below are used when we need to recreate a model frame.

⟨*coxph-finish*⟩=
```
 if (!is.null(weights) && any(weights!=1)) fit$weights <- weights
 if (multi) {
     fit$transitions <- transitions
     fit$states <- states
     fit$cmap <- cmap
     fit$stratum_map <- stratum_map   # why not 'stratamap'?  Confusion with fit$strata
     fit$resid <- rowsum(fit$resid, xstack$rindex)
     # add a suffix to each coefficent name.  Those that map to multiple transitions
     #  get the first transition they map to
     single <- apply(cmap, 1, function(x) all(x %in% c(0, max(x)))) #only 1 coef
     cindx <- col(cmap)[match(1:length(fit$coefficients), cmap)]
     rindx <- row(cmap)[match(1:length(fit$coefficients), cmap)]
     suffix <- ifelse(single[rindx], "", paste0("_", colnames(cmap)[cindx]))
     names(fit$coefficients) <- paste0(names(fit$coefficients), suffix)
     if (x) fit$strata <- istrat  # save the expanded strata
     class(fit) <- c("coxphms", class(fit))
 }
 names(fit$means) <- names(fit$coefficients)

 fit$formula <- formula(Terms)
 if (length(xlevels) >0) fit$xlevels <- xlevels
 fit$contrasts <- contr.save
 if (any(offset !=0)) fit$offset <- offset

 fit$call <- Call
 fit
```

The model.matrix and model.frame routines are called after a Cox model to reconstruct those portions. Much of their code is shared with the coxph routine.

⟨*model.matrix.coxph*⟩=
```
 # In internal use "data" will often be an already derived model frame.
 #  We detect this via it having a terms attribute.
 model.matrix.coxph <- function(object, data=NULL,
                                contrast.arg=object$contrasts, ...) {
     #
     # If the object has an "x" component, return it, unless a new
     #   data set is given
     if (is.null(data) && !is.null(object[['x']]))
         return(object[['x']]) #don't match "xlevels"

     Terms <- delete.response(object$terms)
     if (is.null(data)) mf <- stats::model.frame(object)
```

```
    else {
        if (is.null(attr(data, "terms")))
            mf <- stats::model.frame(Terms, data, xlev=object$xlevels)
        else mf <- data  #assume "data" is already a model frame
    }

    cluster <- attr(Terms, "specials")$cluster
    if (length(cluster)) {
        temp <- untangle.specials(Terms, "cluster")
        dropterms <- temp$terms
    }
    else dropterms <- NULL

    strats <- attr(Terms, "specials")$strata
    hasinteractions <- FALSE
    if (length(strats)) {
        stemp <- untangle.specials(Terms, 'strata', 1)
        if (length(stemp$vars)==1) strata.keep <- mf[[stemp$vars]]
        else strata.keep <- strata(mf[,stemp$vars], shortlabel=TRUE)
        istrat <- as.integer(strata.keep)

        for (i in stemp$vars) {  #multiple strata terms are allowed
            # The factors attr has one row for each variable in the frame, one
            #   col for each term in the model.  Pick rows for each strata
            #   var, and find if it participates in any interactions.
            if (any(attr(Terms, 'order')[attr(Terms, "factors")[i,] >0] >1))
                hasinteractions <- TRUE
        }
        if (!hasinteractions) dropterms <- c(dropterms, stemp$terms)
    } else istrat <- NULL

    ⟨coxph-make-X⟩
    X
}
```

In parallel is the model.frame routine, which reconstructs the model frame. This routine currently doesn't do all that we want. To wit, the following code fails:

```
> tfun <- function(formula, ndata) {
      fit <- coxph(formula, data=ndata)
      model.frame(fit)
      }
> tfun(Surv(time, status) ~ age, lung)
Error: ndata not found
```

The genesis of this problem is hard to unearth, but has to do with non standard evaluation rules used by model.frame.default. In essence it pays attention to the environment of the formula, but

the enclos argument of eval appears to be ignored. I've not yet found a solution.

⟨*model.matrix.coxph*⟩=

```
 model.frame.coxph <- function(formula, ...) {
     dots <- list(...)
     nargs <- dots[match(c("data", "na.action", "subset", "weights",
                           "id", "cluster", "istate"),
                         names(dots), 0)]
     # If nothing has changed and the coxph object had a model component,
     #   simply return it.
     if (length(nargs) ==0  && !is.null(formula$model)) return(formula$model)
     else {
         # Rebuild the original call to model.frame
         Terms <- terms(formula)
         fcall <- formula$call
         indx <- match(c("formula", "data", "weights", "subset", "na.action",
                         "cluster", "id", "istate"),
                   names(fcall), nomatch=0)
         if (indx[1] ==0) stop("The coxph call is missing a formula!")

         temp <- fcall[c(1,indx)]  # only keep the arguments we wanted
         temp[[1]] <- quote(stats::model.frame)  # change the function called
         temp$xlev <- formula$xlevels  # this will turn strings to factors
         temp$formula <- Terms   #keep the predvars attribute
         # Now, any arguments that were on this call overtake the ones that
         #  were in the original call.
         if (length(nargs) >0)
             temp[names(nargs)] <- nargs

         # Make "tt" visible for coxph formulas,
         if (!is.null(attr(temp$formula, "specials")$tt)) {
             coxenv <- new.env(parent= environment(temp$formula))
             assign("tt", function(x) x, envir=coxenv)
             environment(temp$formula) <- coxenv
         }

         # The documentation for model.frame implies that the environment arg
         #  to eval will be ignored, but if we omit it there is a problem.
         if (is.null(environment(formula$terms)))
             mf <- eval(temp, parent.frame())
         else mf <- eval(temp, environment(formula$terms), parent.frame())

         if (!is.null(attr(formula$terms, "dataClasses")))
             .checkMFClasses(attr(formula$terms, "dataClasses"), mf)

         if (is.null(attr(Terms, "specials")$tt)) return(mf)
```

```
        else {
            # Do time transform
            tt <- eval(formula$call$tt)
            Y <- aeqSurv(model.response(mf))
            strats <- attr(Terms, "specials")$strata
            if (length(strats)) {
                stemp <- untangle.specials(Terms, 'strata', 1)
                if (length(stemp$vars)==1) strata.keep <- mf[[stemp$vars]]
                else strata.keep <- strata(mf[,stemp$vars], shortlabel=TRUE)
                istrat <- as.numeric(strata.keep)
            }

            ⟨coxph-transform⟩
            mf[[".strata."]] <- istrat
            return(mf)
        }
    }
}
```

2.2 Exact partial likelihood

Let $r_i = \exp(X_i\beta)$ be the risk score for observation i. For one of the time points assume that there that there are d tied deaths among n subjects at risk. For convenience we will index them as $i = 1, \ldots, d$ in the n at risk. Then for the exact parial likelihood, the contribution at this time point is

$$\begin{aligned} L &= \sum_{i=1}^{d} \log(r_i) - \log(D) \\ \frac{\partial L}{\partial \beta_j} &= x_{ij} - (1/D)\frac{\partial D}{\partial \beta_j} \\ \frac{\partial^2 L}{\partial \beta_j \partial \beta_k} &= (1/D^2)\left[D\frac{\partial^2 D}{\partial \beta_j \partial \beta_k} - \frac{\partial D}{\partial \beta_j}\frac{\partial D}{\partial \beta_k}\right] \end{aligned}$$

The hard part of this computation is D, which is a sum

$$D = \sum_{S(d,n)} r_{s_1} r_{s_2} \ldots r_{s_d}$$

where $S(d, n)$ is the set of all possible subsets of size d from n objects, and $s_1, s_2, \ldots$ indexes the current selection. So if $n = 6$ and $d = 2$ we would have the 15 pairs 12, 13, ... 56; for $n = 5$ and $d = 3$ there would be 10 triples 123, 124, 125, ..., 345.

The brute force computation of all subsets can take a very long time. Gail et al [**?**] show simple recursion formulas that speed this up considerably. Let $D(d, n)$ be the denominator with

d deaths and n subjects. Then

$$D(d,n) = r_n D(d-1,n-1) + D(d,n-1) \tag{1}$$

$$\frac{\partial D(d,n)}{\partial \beta_j} = \frac{\partial D(d,n-1)}{\partial \beta_j} + r_n \frac{\partial D(d-1,n-1)}{\partial \beta_j} + x_{nj} r_n D(d-1,n-1) \tag{2}$$

$$\frac{\partial^2 D(d,n}{\partial \beta_j \partial \beta_k} = \frac{\partial^2 D(d,n-1)}{\partial \beta_j \partial \beta_k} + r_n \frac{\partial^2 D(d-1,n-1)}{\partial \beta_j \partial \beta_k} + x_{nj} r_n \frac{\partial D(d-1,n-1)}{\partial \beta_k} + x_{nk} r_n \frac{\partial D(d-1,n-1)}{\partial \beta_j} + x_{nj} x_{nk} r_n D(d-1,n-1) \tag{3}$$

The above recursion is captured in the three routines below. The first calculates D. It is called with d, n, an array that will contain all the values of $D(d,n)$ computed so far, and the the first dimension of the array. The intial condition $D(0,n) = 1$ is important to all three routines.

⟨*excox-recur*⟩=
```
 double coxd0(int d, int n, double *score, double *dmat,
              int dmax) {
     double *dn;

     if (d==0) return(1.0);
     dn = dmat + (n-1)*dmax + d -1;  /* pointer to dmat[d,n] */

     if (*dn ==0) {  /* still to be computed */
         *dn = score[n-1]* coxd0(d-1, n-1, score, dmat, dmax);
             if (d<n) *dn += coxd0(d, n-1, score, dmat, dmax);
     }
     return(*dn);
 }
```

The next routine calculates the derivative with respect to a particular coefficient. It will be called once for each covariate with d1 pointing to the work array for that covariate. The second derivative calculation is per pair of variables; the `d1j` and `d1k` arrays are the appropriate first derivative arrays of saved values. It is possible for the first derivative to be exactly 0 (if all values of the covariate are 0 for instance) in which case we may recalculate the derivative for a particular (d,n) case multiple times unnecessarily, since we are using value=0 as a marker for "not yet computed". This case is essentially nonexistent in real data, however.

⟨*excox-recur*⟩=
```
 double coxd1(int d, int n, double *score, double *dmat, double *d1,
              double *covar, int dmax) {
     int indx;

     indx = (n-1)*dmax + d -1;  /*index to the current array member d1[d.n]*/
     if (d1[indx] ==0) { /* still to be computed */
         d1[indx] = score[n-1]* covar[n-1]* coxd0(d-1, n-1, score, dmat, dmax);
         if (d<n) d1[indx] += coxd1(d, n-1, score, dmat, d1, covar, dmax);
```

```
        if (d>1) d1[indx] += score[n-1]*
                         coxd1(d-1, n-1, score, dmat, d1, covar, dmax);
    }
    return(d1[indx]);
}

double coxd2(int d, int n, double *score, double *dmat, double *d1j,
             double *d1k, double *d2, double *covarj, double *covark,
             int dmax) {
    int indx;

    indx = (n-1)*dmax + d -1;  /*index to the current array member d1[d,n]*/
    if (d2[indx] ==0) { /*still to be computed */
        d2[indx] = coxd0(d-1, n-1, score, dmat, dmax)*score[n-1] *
            covarj[n-1]* covark[n-1];
        if (d<n) d2[indx] += coxd2(d, n-1, score, dmat, d1j, d1k, d2, covarj,
                                   covark, dmax);
        if (d>1) d2[indx] += score[n-1] * (
            coxd2(d-1, n-1, score, dmat, d1j, d1k, d2, covarj, covark, dmax) +
            covarj[n-1] * coxd1(d-1, n-1, score, dmat, d1k, covark, dmax) +
            covark[n-1] * coxd1(d-1, n-1, score, dmat, d1j, covarj, dmax));
        }
    return(d2[indx]);
}
```

Now for the main body. Start with the dull part of the code: declarations. I use `maxiter2` for the S structure and `maxiter` for the variable within it, and etc for the other input arguments. All the input arguments except strata are read-only. The output beta vector starts as a copy of ibeta.

⟨*coxexact*⟩=

```
#include <math.h>
#include "survS.h"
#include "survproto.h"
#include <R_ext/Utils.h>

⟨excox-recur⟩

SEXP coxexact(SEXP maxiter2,  SEXP y2,
              SEXP covar2,    SEXP offset2, SEXP strata2,
              SEXP ibeta,     SEXP eps2,    SEXP toler2) {
    int i,j,k;
    int     iter;

    double **covar, **imat;  /*ragged arrays */
    double *time, *status;   /* input data */
    double *offset;
```

```
    int    *strata;
    int    sstart;   /* starting obs of current strata */
    double *score;
    double *oldbeta;
    double  zbeta;
    double  newlk=0;
    double  temp;
    int     halving;    /*are we doing step halving at the moment? */
    int     nrisk =0;   /* number of subjects in the current risk set */
    int dsize,       /* memory needed for one coxc0, coxc1, or coxd2 array */
        dmemtot,     /* amount needed for all arrays */
        ndeath;      /* number of deaths at the current time point */
    double maxdeath;    /* max tied deaths within a strata */

    double dtime;    /* time value under current examiniation */
    double *dmem0, **dmem1, *dmem2; /* pointers to memory */
    double *dtemp;   /* used for zeroing the memory */
    double *d1;     /* current first derivatives from coxd1 */
    double d0;      /* global sum from coxc0 */

    /* copies of scalar input arguments */
    int     nused, nvar, maxiter;
    double  eps, toler;

    /* returned objects */
    SEXP imat2, beta2, u2, loglik2;
    double *beta, *u, *loglik;
    SEXP rlist, rlistnames;
    int nprotect;  /* number of protect calls I have issued */

    ⟨excox-setup⟩
    ⟨excox-strata⟩
    ⟨excox-iter0⟩
    ⟨excox-iter⟩
    }
```

Setup is ordinary. Grab S objects and assign others. I use `R_alloc` for temporary ones since it is released automatically on return.

```
⟨excox-setup⟩=
 nused = LENGTH(offset2);
 nvar  = ncols(covar2);
 maxiter = asInteger(maxiter2);
 eps  = asReal(eps2);     /* convergence criteria */
 toler = asReal(toler2);  /* tolerance for cholesky */

 /*
```

```
**  Set up the ragged array pointer to the X matrix,
**    and pointers to time and status
*/
covar= dmatrix(REAL(covar2), nused, nvar);
time = REAL(y2);
status = time +nused;
strata = INTEGER(PROTECT(duplicate(strata2)));
offset = REAL(offset2);

/* temporary vectors */
score = (double *) R_alloc(nused+nvar, sizeof(double));
oldbeta = score + nused;

/*
** create output variables
*/
PROTECT(beta2 = duplicate(ibeta));
beta = REAL(beta2);
PROTECT(u2 = allocVector(REALSXP, nvar));
u = REAL(u2);
PROTECT(imat2 = allocVector(REALSXP, nvar*nvar));
imat = dmatrix(REAL(imat2),  nvar, nvar);
PROTECT(loglik2 = allocVector(REALSXP, 5)); /* loglik, sctest, flag,maxiter*/
loglik = REAL(loglik2);
nprotect = 5;
```

The data passed to us has been sorted by strata, and reverse time within strata (longest subject first). The variable `strata` will be 1 at the start of each new strata. Separate strata are completely separate computations: time 10 in one strata and time 10 in another are not comingled. Compute the largest product (size of strata)* (max tied deaths in strata) for allocating scratch space. When computing D it is advantageous to create all the intermediate values of $D(d, n)$ in an array since they will be used in the derivative calculation. Likewise, the first derivatives are used in calculating the second. Even more importantly, say we have a large data set. It will be sorted with the longest times first. If there is a death with 30 at risk and another with 40 at risk, the intermediate sums we computed for the n=30 case are part of the computation for n=40. To make this work we need to index our matrices, within any strata, by the maximum number of tied deaths in the strata. We save this in the strata variable: first obs of a new strata has the number of events. And what if a strata had 0 events? We mark it with a 1.

Note that the maxdeath variable is floating point. I had someone call this routine with a data set that gives an integer overflow in that situation. We now keep track of this further below and fail with a message. Such a run would take longer than forever to complete even if integer subscripts did not overflow.

⟨*excox-strata*⟩=
```
strata[0] =1;  /* in case the parent forgot (e.g., no strata case)*/
temp = 0;      /* temp variable for dsize */
```

```
maxdeath =0;
j=0;   /* start of the strata */
for (i=0; i<nused;) {
    if (strata[i]==1) { /* first obs of a new strata */
       if (i>0) {
           /* assign data for the prior stratum, just finished */
           /* If maxdeath <2 leave the strata alone at it's current value of 1 */
           if (maxdeath >1) strata[j] = maxdeath;
           j = i;
           if (maxdeath*nrisk > temp) temp = maxdeath*nrisk;
       }
       maxdeath =0;  /* max tied deaths at any time in this strata */
       nrisk=0;
       ndeath =0;
    }
    dtime = time[i];
    ndeath =0;  /*number tied here */
    while (time[i] ==dtime) {
        nrisk++;
        ndeath += status[i];
        i++;
        if (i>=nused || strata[i] >0) break;  /*tied deaths don't cross strata */
    }
    if (ndeath > maxdeath) maxdeath = ndeath;
}
/* data for the final stratum */
if (maxdeath*nrisk > temp) temp = maxdeath*nrisk;
if (maxdeath >1) strata[j] = maxdeath;

/* Now allocate memory for the scratch arrays
   Each per-variable slice is of size dsize
*/
dsize = temp;
temp    = temp * ((nvar*(nvar+1))/2 + nvar + 1);
dmemtot = dsize * ((nvar*(nvar+1))/2 + nvar + 1);
if (temp != dmemtot) { /* the subscripts will overflow */
    error("(number at risk) * (number tied deaths) is too large");
}
dmem0 = (double *) R_alloc(dmemtot, sizeof(double)); /*pointer to memory */
dmem1 = (double **) R_alloc(nvar, sizeof(double*));
dmem1[0] = dmem0 + dsize; /*points to the first derivative memory */
for (i=1; i<nvar; i++) dmem1[i] = dmem1[i-1] + dsize;
d1 = (double *) R_alloc(nvar, sizeof(double)); /*first deriv results */
```

Here is a standard iteration step. Walk forward to a new time, then through all the ties

with that time. If there are any deaths, the contributions to the loglikilihood, first, and second derivatives at this time point are

$$L = \left(\sum_{i \in deaths} X_i \beta \right) - \log(D) \tag{4}$$

$$\frac{\partial L}{\partial \beta_j} = \left(\sum_{i \in deaths} X_{ij} \right) - \frac{\partial D(d,n)}{\partial \beta_j} D^{-1}(d,n) \tag{5}$$

$$\frac{\partial^2 L}{\partial \beta_j \partial \beta_k} = \frac{\partial^2 D(d,n)}{\partial \beta_j \partial \beta_k} D^{-1}(d,n) - \frac{\partial D(d,n)}{\partial \beta_j} \frac{\partial D(d,n)}{\partial \beta_k} D^{-2}(d,n) \tag{6}$$

Even the efficient calculation can be compuatationally intense, so check for user interrupt requests on a regular basis.

⟨ *excox-addup* ⟩=

```
 sstart =0;  /* a line to make gcc stop complaining */
 for (i=0; i<nused; ) {
     if (strata[i] >0) { /* first obs of a new strata */
         maxdeath= strata[i];
         dtemp = dmem0;
         for (j=0; j<dmemtot; j++) *dtemp++ =0.0;
         sstart =i;
         nrisk =0;
     }

     dtime = time[i];  /*current unique time */
     ndeath =0;
     while (time[i] == dtime) {
         zbeta= offset[i];
         for (j=0; j<nvar; j++) zbeta += covar[j][i] * beta[j];
         score[i] = exp(zbeta);
         if (status[i]==1) {
             newlk += zbeta;
             for (j=0; j<nvar; j++) u[j] += covar[j][i];
             ndeath++;
         }
         nrisk++;
         i++;
         if (i>=nused || strata[i] >0) break;
     }

     /* We have added up over the death time, now process it */
     if (ndeath >0) { /* Add to the loglik */
         d0 = coxd0(ndeath, nrisk, score+sstart, dmem0, maxdeath);
         R_CheckUserInterrupt();
         newlk -= log(d0);
```

```
        dmem2 = dmem0 + (nvar+1)*dsize;  /*start for the second deriv memory */
        for (j=0; j<nvar; j++) { /* for each covariate */
            d1[j] = coxd1(ndeath, nrisk, score+sstart, dmem0, dmem1[j],
                          covar[j]+sstart, maxdeath) / d0;
            if (ndeath > 3) R_CheckUserInterrupt();
            u[j] -= d1[j];
            for (k=0; k<= j; k++) {  /* second derivative*/
                temp = coxd2(ndeath, nrisk, score+sstart, dmem0, dmem1[j],
                             dmem1[k], dmem2, covar[j] + sstart,
                             covar[k] + sstart, maxdeath);
                if (ndeath > 5) R_CheckUserInterrupt();
                imat[k][j] += temp/d0 - d1[j]*d1[k];
                dmem2 += dsize;
            }
        }
    }
}
```

Do the first iteration of the solution. The first iteration is different in 3 ways: it is used to set the initial log-likelihood, to compute the score test, and we pay no attention to convergence criteria or diagnositics. (I expect it not to converge in one iteration).

⟨*excox-iter0*⟩=

```
/*
** do the initial iteration step
*/
newlk =0;
for (i=0; i<nvar; i++) {
    u[i] =0;
    for (j=0; j<nvar; j++)
        imat[i][j] =0 ;
    }
⟨excox-addup⟩

loglik[0] = newlk;   /* save the loglik for iteration zero  */
loglik[1] = newlk;  /* and it is our current best guess */
/*
**   update the betas and compute the score test
*/
for (i=0; i<nvar; i++) /*use 'd1' as a temp to save u0, for the score test*/
    d1[i] = u[i];

loglik[3] = cholesky2(imat, nvar, toler);
chsolve2(imat,nvar, u);        /* u replaced by  u *inverse(imat) */

loglik[2] =0;                  /* score test stored here */
for (i=0; i<nvar; i++)
```

```
    loglik[2] +=  u[i]*d1[i];

 if (maxiter==0) {
    iter =0;  /*number of iterations */
    ⟨excox-finish⟩
    }

 /*
 **  Never, never complain about convergence on the first step.  That way,
 **  if someone has to they can force one iter at a time.
 */
 for (i=0; i<nvar; i++) {
    oldbeta[i] = beta[i];
    beta[i] = beta[i] + u[i];
    }
```

Now the main loop. This has code for convergence and step halving. Be careful about order. For our current guess at the solution beta:

1. Compute the loglik, first, and second derivatives
2. If the loglik has converged, return beta and information just computed for this beta (loglik, derivatives, etc). Don't update beta.
3. If not converged
   - If The loglik got worse try beta= (beta + oldbeta)/2
   - Otherwise update beta

```
⟨excox-iter⟩=
 halving =0 ;             /* =1 when in the midst of "step halving" */
 for (iter=1; iter<=maxiter; iter++) {
    newlk =0;
    for (i=0; i<nvar; i++) {
        u[i] =0;
        for (j=0; j<nvar; j++)
                imat[i][j] =0;
        }
    ⟨excox-addup⟩

    /* am I done?
    **   update the betas and test for convergence
    */
    loglik[3] = cholesky2(imat, nvar, toler);

    if (fabs(1-(loglik[1]/newlk))<= eps && halving==0) { /* all done */
        loglik[1] = newlk;
```

```
    ⟨excox-finish⟩
     }

  if (iter==maxiter) break;  /*skip the step halving and etc */

  if (newlk < loglik[1])   {    /*it is not converging ! */
          halving =1;
          for (i=0; i<nvar; i++)
              beta[i] = (oldbeta[i] + beta[i]) /2; /*half of old increment */
          }
  else {
          halving=0;
          loglik[1] = newlk;
          chsolve2(imat,nvar,u);

          for (i=0; i<nvar; i++) {
              oldbeta[i] = beta[i];
              beta[i] = beta[i] +  u[i];
              }
          }
  }   /* return for another iteration */


 /*
 ** Ran out of iterations
 */
 loglik[1] = newlk;
 loglik[3] = 1000;  /* signal no convergence */
 ⟨excox-finish⟩
```

The common code for finishing. Invert the information matrix, copy it to be symmetric, and put together the output structure.

```
⟨excox-finish⟩=
 loglik[4] = iter;
 chinv2(imat, nvar);
 for (i=1; i<nvar; i++)
     for (j=0; j<i; j++)  imat[i][j] = imat[j][i];

 /* assemble the return objects as a list */
 PROTECT(rlist= allocVector(VECSXP, 4));
 SET_VECTOR_ELT(rlist, 0, beta2);
 SET_VECTOR_ELT(rlist, 1, u2);
 SET_VECTOR_ELT(rlist, 2, imat2);
 SET_VECTOR_ELT(rlist, 3, loglik2);
```

```
/* add names to the list elements */
PROTECT(rlistnames = allocVector(STRSXP, 4));
SET_STRING_ELT(rlistnames, 0, mkChar("coef"));
SET_STRING_ELT(rlistnames, 1, mkChar("u"));
SET_STRING_ELT(rlistnames, 2, mkChar("imat"));
SET_STRING_ELT(rlistnames, 3, mkChar("loglik"));
setAttrib(rlist, R_NamesSymbol, rlistnames);

unprotect(nprotect+2);
return(rlist);
```

2.3 Andersen-Gill fits

When the survival data set has (start, stop] data a couple of computational issues are added. A primary one is how to do this compuation efficiently. At each event time we need to compute 3 quantities, each of them added up over the current risk set.

- The weighted sum of the risk scores $\sum w_i r_i$ where $r_i = \exp(\eta_i)$ and $\eta_i = x_{i1}\beta_1 + x_{i2}\beta_2 + \ldots$ is the current linear predictor.
- The weighted mean of the covariates x, with weight $w_i r_i$.
- The weighted variance-covariance matrix of x.

The current risk set at some event time t is the set of all (start, stop] intervals that overlap t, and are part of the same strata. The round/square brackets in the prior sentence are important: for an event time $t = 20$ the interval $(5, 20]$ is considered to overlap t and the interval $(20, 55]$ does not overlap t.

Our routine for the simple right censored Cox model computes these efficiently by keeping a cumulative sum. Starting with the longest survival move backwards through time, adding and subtracting subject from the sum as we go. The code below creates two sort indices, one orders the data by reverse stop time and the other by reverse start time, each within strata.

The fit routine is called by the coxph function with arguments

x matrix of covariates

y three column matrix containing the start time, stop time, and event for each observation

strata for stratified fits, the strata of each subject

offset the offset, usually a vector of zeros

init initial estimate for the coefficients

control results of the coxph.control function

weights case weights, often a vector of ones.

method how ties are handled: 1=Breslow, 2=Efron

rownames used to label the residuals

If the data set has any observations whose (start, stop] interval does not overlap any death times, those rows of data play no role in the computation, and we push them to the end of the sort order and report a smaller n to the C routine. The reason for this has less to do with efficiency than with safety: one user, for example, created a data set with a time*covariate interaction, to be used for testing proportional hazards with an `x:ns(time, df=4)` term. They had cut the data up by day using survSplit, there was a long no-event stretch of time before the last censor, and this generated some large outliers in the extrapolated spline — large enough to force an exp() overflow.

⟨*agreg.fit*⟩=
```
 agreg.fit <- function(x, y, strata, offset, init, control,
                       weights, method, rownames, resid=TRUE, nocenter=NULL)
     {
     nvar <- ncol(x)
     event <- y[,3]
     if (all(event==0)) stop("Can't fit a Cox model with 0 failures")

     if (missing(offset) || is.null(offset)) offset <- rep(0.0, nrow(y))
     if (missing(weights)|| is.null(weights))weights<- rep(1.0, nrow(y))
     else if (any(weights<=0)) stop("Invalid weights, must be >0")
     else weights <- as.vector(weights)

     # Find rows to be ignored.  We have to match within strata: a
     #  value that spans a death in another stratum, but not it its
     #  own, should be removed.  Hence the per stratum delta
     if (length(strata) ==0) {y1 <- y[,1]; y2 <- y[,2]}
     else  {
         if (is.numeric(strata)) strata <- as.integer(strata)
         else strata <- as.integer(as.factor(strata))
         delta  <-  strata* (1+ max(y[,2]) - min(y[,1]))
         y1 <- y[,1] + delta
         y2 <- y[,2] + delta
     }
     event <- y[,3] > 0
     dtime <- sort(unique(y2[event]))
     indx1 <- findInterval(y1, dtime)
     indx2 <- findInterval(y2, dtime)
     # indx1 != indx2 for any obs that spans an event time
     ignore <- (indx1 == indx2)
     nused  <- sum(!ignore)

     # Sort the data (or rather, get a list of sorted indices)
     #  For both stop and start times, the indices go from last to first
     if (length(strata)==0) {
         sort.end  <- order(ignore, -y[,2]) -1L #indices start at 0 for C code
         sort.start<- order(ignore, -y[,1]) -1L
```

```
    strata <- rep(0L, nrow(y))
    }
else {
    sort.end  <- order(ignore, strata, -y[,2]) -1L
    sort.start<- order(ignore, strata, -y[,1]) -1L
    }

if (is.null(nvar) || nvar==0) {
    # A special case: Null model.  Just return obvious stuff
    #  To keep the C code to a small set, we call the usual routines, but
    #  with a dummy X matrix and 0 iterations
    nvar <- 1
    x <- matrix(as.double(1:nrow(y)), ncol=1)  #keep the .C call happy
    maxiter <- 0
    nullmodel <- TRUE
    if (length(init) !=0) stop("Wrong length for inital values")
    init <- 0.0  #dummy value to keep a .C call happy (doesn't like 0 length)
    }
else {
    nullmodel <- FALSE
    maxiter <- control$iter.max

    if (is.null(init)) init <- rep(0., nvar)
    if (length(init) != nvar) stop("Wrong length for inital values")
    }

# 2021 change: pass in per covariate centering.  This gives
#  us more freedom to experiment.  Default is to leave 0/1 variables alone
if (is.null(nocenter)) zero.one <- rep(FALSE, ncol(x))
zero.one <- apply(x, 2, function(z) all(z %in% nocenter))

# the returned value of agfit$coef starts as a copy of init, so make sure
#  is is a vector and not a matrix; as.double suffices.
# Solidify the storage mode of other arguments
storage.mode(y) <- storage.mode(x) <- "double"
storage.mode(offset) <- storage.mode(weights) <- "double"
agfit <- .Call(Cagfit4, nused,
               y, x, strata, weights,
               offset,
               as.double(init),
               sort.start, sort.end,
               as.integer(method=="efron"),
               as.integer(maxiter),
               as.double(control$eps),
               as.double(control$toler.chol),
               ifelse(zero.one, 0L, 1L))
```

```
    # agfit4 centers variables within strata, so does not return a vector
    #  of means.  Use a fill in consistent with other coxph routines
    agmeans <- ifelse(zero.one, 0, colMeans(x))

    ⟨agreg-fixup⟩
    ⟨agreg-finish⟩
    rval
}
```

Upon return we need to clean up three simple things. The first is the rare case that the agfit routine failed. These cases are rare, usually involve an overflow or underflow, and we encourage users to let us have a copy of the data when it occurs. (They end up in the `fail` directory of the library.) The second is that if any of the covariates were redudant then this will be marked by zeros on the diagonal of the variance matrix. Replace these coefficients and their variances with NA. The last is to post a warning message about possible infinite coefficients. The algorithm for determining this is unreliable, unfortunately. Sometimes coefficients are marked as infinite when the solution is not tending to infinity (usually associated with a very skewed covariate), and sometimes one that is tending to infinity is not marked. Que sera sera. Don't complain if the user asked for only one iteration; they will already know that it has not converged.

```
⟨agreg-fixup⟩=
 vmat <- agfit$imat
 coef <- agfit$coef
 if (agfit$flag[1] < nvar) which.sing <- diag(vmat)==0
 else which.sing <- rep(FALSE,nvar)

 if (maxiter >1) {
     infs <- abs(agfit$u %*% vmat)
     if (any(!is.finite(coef)) || any(!is.finite(vmat)))
         stop("routine failed due to numeric overflow.",
              "This should never happen.  Please contact the author.")
     if (agfit$flag[4] > 0)
         warning("Ran out of iterations and did not converge")
     else {
         infs <- (!is.finite(agfit$u) |
                  infs > control$toler.inf*(1+ abs(coef)))
         if (any(infs))
             warning(paste("Loglik converged before variable ",
                           paste((1:nvar)[infs],collapse=","),
                           "; beta may be infinite. "))
     }
 }
```

The last of the code is very standard. Compute residuals and package up the results. One design decision is that we return all n residuals and predicted values, even though the model fit ignored useless observations. (All those obs have a residual of 0).

⟨*agreg-finish*⟩=
```
lp  <- as.vector(x %*% coef + offset - sum(coef * agmeans))
if (resid) {
    if (any(lp > log(.Machine$double.xmax))) {
        # prevent a failure message due to overflow
        #  this occurs with near-infinite coefficients
        temp <- lp + log(.Machine$double.xmax) - (1 + max(lp))
        score <- exp(temp)
    } else score <- exp(lp)

    residuals <- .Call(Cagmart3, nused,
                   y, score, weights,
                   strata,
                   sort.start, sort.end,
                   as.integer(method=='efron'))
    names(residuals) <- rownames
}

# The if-then-else below is a real pain in the butt, but the tccox
#  package's test suite assumes that the ORDER of elements in a coxph
#  object will never change.
#
if (nullmodel) {
    rval <- list(loglik=agfit$loglik[2],
         linear.predictors = offset,
         method= method,
         class = c("coxph.null", 'coxph') )
    if (resid) rval$residuals <- residuals
}
else {
    names(coef) <- dimnames(x)[[2]]
    if (maxiter > 0) coef[which.sing] <- NA  # always leave iter=0 alone
    flag <- agfit$flag
    names(flag) <- c("rank", "rescale", "step halving", "convergence")

    if (resid) {
        rval <- list(coefficients  = coef,
                     var    = vmat,
                     loglik = agfit$loglik,
                     score  = agfit$sctest,
                     iter   = agfit$iter,
                     linear.predictors = as.vector(lp),
                     residuals = residuals,
                     means = agmeans,
                     first = agfit$u,
                     info = flag,
```

```
                    method= method,
                    class = "coxph")
    } else {
        rval <- list(coefficients  = coef,
                    var    = vmat,
                    loglik = agfit$loglik,
                    score  = agfit$sctest,
                    iter   = agfit$iter,
                    linear.predictors = as.vector(lp),
                    means = agmeans,
                    first = agfit$u,
                    info = flag,
                    method = method,
                    class = "coxph")
    }
    rval
}
```

The details of the C code contain the more challenging part of the computations. It starts with the usual dull stuff. My standard coding style for a variable zed to to use `zed2` as the variable name for the R object, and `zed` for the pointer to the contents of the object, i.e., what the C code will manipulate. For the matrix objects I make use of ragged arrays, this allows for reference to the i,j element as `cmat[i][j]` and makes for more readable code.

⟨*agfit4*⟩=

```
#include <math.h>
#include "survS.h"
#include "survproto.h"

SEXP agfit4(SEXP nused2, SEXP surv2,      SEXP covar2,    SEXP strata2,
            SEXP weights2,   SEXP offset2,   SEXP ibeta2,
            SEXP sort12,     SEXP sort22,    SEXP method2,
            SEXP maxiter2,   SEXP  eps2,     SEXP tolerance2,
            SEXP doscale2) {

    int i,j,k, person;
    int indx1, istrat, p, p1;
    int nrisk, nr;
    int nused, nvar;
    int rank=0, rank2, fail;  /* =0 to keep -Wall happy */

    double **covar, **cmat, **imat;  /*ragged array versions*/
    double *a, *oldbeta;
    double *scale;
    double *a2, **cmat2;
    double *eta;
    double  denom, zbeta, risk;
```

```
double  dtime =0;  /* initial value to stop a -Wall message */
double  temp, temp2;
double  newlk =0;
int  halving;    /*are we doing step halving at the moment? */
double  tol_chol, eps;
double  meanwt;
int deaths;
double denom2, etasum;
double recenter;

/* inputs */
double *start, *tstop, *event;
double *weights, *offset;
int *sort1, *sort2, maxiter;
int *strata;
double method;  /* saving this as double forces some double arithmetic */
int *doscale;

/* returned objects */
SEXP imat2, beta2, u2, loglik2;
double *beta, *u, *loglik;
SEXP sctest2, flag2, iter2;
double *sctest;
int *flag, *iter;
SEXP rlist;
static const char *outnames[]={"coef", "u", "imat", "loglik",
                               "sctest", "flag", "iter", ""};
int nprotect;  /* number of protect calls I have issued */

/* get sizes and constants */
nused = asInteger(nused2);
nvar  = ncols(covar2);
nr    = nrows(covar2);  /*nr = number of rows, nused = how many we use */
method= asInteger(method2);
eps   = asReal(eps2);
tol_chol = asReal(tolerance2);
maxiter = asInteger(maxiter2);
doscale = INTEGER(doscale2);

/* input arguments */
start = REAL(surv2);
tstop  = start + nr;
event = tstop + nr;
weights = REAL(weights2);
offset = REAL(offset2);
sort1  = INTEGER(sort12);
```

```
sort2  = INTEGER(sort22);
strata = INTEGER(strata2);

/*
** scratch space
**  nvar: a, a2, oldbeta, scale
**  nvar*nvar: cmat, cmat2
**  nr:  eta
*/
eta = (double *) R_alloc(nr + 4*nvar + 2*nvar*nvar, sizeof(double));
a = eta + nr;
a2= a + nvar;
scale  = a2 + nvar;
oldbeta = scale + nvar;

/*
**  Set up the ragged arrays
**  covar2 might not need to be duplicated, even though
**  we are going to modify it, due to the way this routine was
**  was called.  But check
*/
PROTECT(imat2 = allocMatrix(REALSXP, nvar, nvar));
nprotect =1;
if (MAYBE_REFERENCED(covar2)) {
    PROTECT(covar2 = duplicate(covar2));
    nprotect++;
    }
covar= dmatrix(REAL(covar2), nr, nvar);
imat = dmatrix(REAL(imat2),  nvar, nvar);
cmat = dmatrix(oldbeta+ nvar,   nvar, nvar);
cmat2= dmatrix(oldbeta+ nvar + nvar*nvar, nvar, nvar);

/*
** create the output structures
*/
PROTECT(rlist = mkNamed(VECSXP, outnames));
nprotect++;
beta2 = SET_VECTOR_ELT(rlist, 0, duplicate(ibeta2));
beta  = REAL(beta2);
u2 =    SET_VECTOR_ELT(rlist, 1, allocVector(REALSXP, nvar));
u = REAL(u2);

SET_VECTOR_ELT(rlist, 2, imat2);
loglik2 = SET_VECTOR_ELT(rlist, 3, allocVector(REALSXP, 2));
loglik  = REAL(loglik2);
```

```
sctest2 = SET_VECTOR_ELT(rlist, 4, allocVector(REALSXP, 1));
sctest =  REAL(sctest2);
flag2  =  SET_VECTOR_ELT(rlist, 5, allocVector(INTSXP, 4));
flag   =  INTEGER(flag2);
for (i=0; i<4; i++) flag[i]=0;

iter2  =  SET_VECTOR_ELT(rlist, 6, allocVector(INTSXP, 1));
iter = INTEGER(iter2);

/*
** Subtract the mean from each covar, as this makes the variance
**  computation more stable.  The mean is taken per stratum,
**  the scaling is overall.
*/
for (i=0; i<nvar; i++) {
    if (doscale[i] == 0) scale[i] =1; /* skip this variable */
    else {
        istrat = strata[sort2[0]];  /* the current stratum */
        k = 0;                      /* first obs of current one */
        temp =0;  temp2=0;
        for (person=0; person< nused; person++) {
            p = sort2[person];
            if (strata[p] == istrat) {
                temp += weights[p] * covar[i][p];
            temp2 += weights[p];
            }
            else {  /* new stratum */
                temp /= temp2;  /* mean for this covariate, this strata */
                for (; k< person; k++) covar[i][sort2[k]] -=temp;
                temp =0;  temp2=0;
                istrat = strata[p];
            }
            temp /= temp2;  /* mean for last stratum */
            for (; k< nused; k++) covar[i][sort2[k]] -= temp;
        }

        /* this cannot be done per stratum */
        temp =0;
        temp2 =0;
        for (person=0; person<nused; person++) {
            p = sort2[person];
            temp += weights[p] * fabs(covar[i][p]);
            temp2 += weights[p];
            }
        if (temp >0) temp = temp2/temp;  /* 1/scale */
        else temp = 1.0;  /* rare case of a constant covariate */
```

```
            scale[i] = temp;
            for (person=0; person<nused; person++) {
                covar[i][sort2[person]] *= temp;
            }
        }
    }

    for (i=0; i<nvar; i++) beta[i] /= scale[i]; /* rescale initial betas */

    ⟨agfit4-iter⟩
    ⟨agfit4-finish⟩
}
```

As we walk through the risk sets observations are both added and removed from a set of running totals. We have 6 running totals:

- sum of the weights, denom = $\sum w_i r_i$
- totals for each covariate a[j] = $\sum w_i r_i x_{ij}$
- totals for each covariate pair cmat[j,k]= $\sum w_i r_i x_{ij} x_{ik}$
- the same three quantities, but only for times that are exactly tied with the current death time, named denom2, a2, cmat2. This allows for easy compuatation of the Efron approximation for ties.

At one point I spent a lot of time worrying about r_i values that are too large, but it turns out that the overall scale of the weights does not really matter since they always appear as a ratio. (Assuming we avoid exponential overflow and underflow, of course.) What does get the code in trouble is when there are large and small weights and we get an update of (large + small) - large. For example suppose a data set has a time dependent covariate which grows with time and the data has values like below:

time1	time2	status	x
0	90	1	1
0	105	0	2
100	120	1	50
100	124	0	51

The code moves from large times to small, so the first risk set has subjects 3 and 4, the second has 1 and 2. The original code would do removals only when necessary, i.e., at the event times of 120 and 90, and additions as they came along. This leads to adding in subjects 1 and 2 before the update at time 90 when observations 3 and 4 are removed; for a coefficient greater than about .6 this leads to a loss of all of the significant digits. The defense is to remove subjects from the risk set as early as possible, and defer additions for as long as possible. Every time we hit a new (unique) death time, and only then, update the totals: first remove any old observations no longer in the risk set and then add any new ones.

One interesting edge case is observations that are not part of any risk set. (A call to survSplit with too fine a partition can create these, or using a subset of data that excluded some of the

deaths.) Observations that are not part of any risk set add unnecessary noise since they will be added and then subtracted from all the totals, but the intermediate values are never used. If said observation had a large risk score this could be exceptionally bad. The parent routine has already dealt with such observations: their indices never appear in the sort1 or sort2 vector.

The three primary quantities for the Cox model are the log-likelihood L, the score vector U and the Hessian matrix H.

$$\begin{aligned}
L &= \sum_i w_i \delta_i \left[\eta_i - \log(d(t))\right] \\
d(t) &= \sum_j w_j r_j Y_j(t) \\
U_k &= \sum_i w_i \delta_i \left[(X_{ik} - \mu_k(t_i))\right] \\
\mu_k(t) &= \frac{\sum_j w_j r_j Y_j(t) X_{jk}}{d(t)} \\
H_{kl} &= \sum_i w_i \delta_i V_{kl}(t_i) \\
V_{kl}(t) &= \frac{\sum_j w_j r_j Y_j(t)[X_{jk} - \mu_k(t)][X_{jl} - \mu_l(t)]}{d(t)} \\
&= \frac{\sum_j w_j r_j Y_j(t) X_{jk} X_{jl}}{d(t)} - d(t)\mu_k(t)\mu_l(t)
\end{aligned}$$

In the above $\delta_i = 1$ for an event and 0 otherwise, w_i is the per subject weight, η_i is the current linear predictor $X\beta$ for the subject, $r_i = \exp(\eta_i)$ is the risk score and $Y_i(t)$ is 1 if observation i is at risk at time t. The vector $\mu(t)$ is the weighted mean of the covariates at time t using a weight of $wrY(t)$ for each subject, and $V(t)$ is the weighted variance matrix of X at time t.

Tied deaths and the Efron approximation add a small complication to the formula. Say there are three tied deaths at some particular time t. When calculating the denominator $d(t)$, mean $\mu(t)$ and variance $V(t)$ at that time the inclusion value $Y_i(t)$ is 0 or 1 for all other subjects, as usual, but for the three tied deaths Y(t) is taken to be 1 for the first death, 2/3 for the second, and 1/3 for the third. The idea is that if the tied death times were randomly broken by adding a small random amount then each of these three would be in the first risk set, have 2/3 chance of being in the second, and 1/3 chance of being in the risk set for the third death. In the code this means that at a death time we add the `denom2`, `a2` and `c2` portions in a little at at time: for three tied death the code will add in 1/3, update totals, add in another 1/3, update totals, then the last 1/3, and update totals.

The variance formula is stable if μ is small relative to the total variance. This is guarranteed by having a working estimate m of the mean along with the formula:

$$\begin{aligned}
(1/n)\sum w_i r_i (x_i - \mu)^2 &= (1/n)\sum w_i r_i (x - m)^2 - (\mu - m)^2 \\
\mu &= (1/n)\sum w_i r_i (x_i - m) \\
n &= \sum w_i r_i
\end{aligned}$$

A refinement of this is to scale the covariates, since the Cholesky decomposition can lose precision when variables are on vastly different scales. We do this centering and scaling once at the beginning of the calculation. Centering is done per strata — what if someone had two strata and a covariate with mean 0 in the first but mean one million in the second? (Users do amazing things). Scaling is required to be a single value for each covariate, however. For a univariate model scaling does not add any precision.

Weighted sums can still be unstable if the weights get out of hand. Because of the exponential $r_i = exp(\eta_i)$ the original centering of the X matrix may not be enough. A particular example was a data set on hospital adverse events with "number of nurse shift changes to date" as a time dependent covariate. At any particular time point the covariate varied only by ± 3 between subjects (weekends often use 12 hour nurse shifts instead of 8 hour). The regression coefficient was around 1 and the data duration was 11 weeks (about 200 shifts) so that *eta* values could be over 100 even after centering. We keep a time dependent average of η and use it to update a recentering constant as necessary. A case like this should be rare, but it is not as unusual as one might think.

The last numerical problem is when one or more coefficients gets too large, leading to a huge weight exp(eta). This usually happens when a coefficient is tending to infinity, but can also be due to a bad step in the intermediate Newton-Raphson path. In the infinite coefficient case the log-likelihood trends to an asymptote and there is a race between three conditions: convergence of the loglik, singularity of the variance matrix, or an invalid log-likelihood. The first of these wins the race most of the time, especially if the data set is small, and is the simplest case. The last occurs when the denominator becomes < 0 due to round off so that log(denom) is undefined, the second when extreme weights cause the second derivative to lose precision. In all 3 we revert to step halving, since a bad Newton-Raphson step can cause the same issues to arise.

The next section of code adds up the totals for a given iteration. This is the workhorse. For a given death time all of the events tied at that time must be handled together, hence the main loop below proceeds in batches:

1. Find the time of the next death. Whenever crossing a stratum boundary, zero cetain intermediate sums.

2. Remove all observations in the stratum with time1 $>$ dtime. When survSplit was used to create a data set, this will often remove all. If so we can rezero temporaries and regain precision.

3. Add new observations to the risk set and to the death counts.

⟨ *agfit4-addup* ⟩=
```
 for (person=0; person<nused; person++) {
     p = sort2[person];
     zbeta = 0;      /* form the term beta*z   (vector mult) */
     for (i=0; i<nvar; i++)
         zbeta += beta[i]*covar[i][p];
     eta[p] = zbeta + offset[p];
 }

 /*
```

```
**  'person' walks through the the data from 1 to nused,
**     sort1[0] points to the largest stop time, sort1[1] the next, ...
**  'dtime' is a scratch variable holding the time of current interest
**  'indx1' walks through the start times.
*/
newlk =0;
for (i=0; i<nvar; i++) {
    u[i] =0;
    for (j=0; j<nvar; j++) imat[i][j] =0;
}
person =0;
indx1 =0;

/* this next set is rezeroed at the start of each stratum */
recenter =0;
denom=0;
nrisk=0;
etasum =0;
for (i=0; i<nvar; i++) {
    a[i] =0;
    for (j=0; j<nvar; j++) cmat[i][j] =0;
}
/* end of the per-stratum set */

istrat = strata[sort2[0]];  /* initial stratum */
while (person < nused) {
    /* find the next death time */
    for (k=person; k< nused; k++) {
        p = sort2[k];
        if (strata[p] != istrat) {
            /* hit a new stratum; reset temporary sums */
            istrat= strata[p];
            denom = 0;
            nrisk = 0;
            etasum =0;
            for (i=0; i<nvar; i++) {
                a[i] =0;
                for (j=0; j<nvar; j++) cmat[i][j] =0;
            }
            person =k;  /* skip to end of stratum */
            indx1  =k;
        }

        if (event[p] == 1) {
            dtime = tstop[p];
            break;
```

```
    }
}
if (k == nused) break;  /* no more deaths to be processed */

/* remove any subjects no longer at risk */
⟨agreg-remove⟩

/*
** add any new subjects who are at risk
** denom2, a2, cmat2, meanwt and deaths count only the deaths
*/
denom2= 0;
meanwt =0;
deaths=0;
for (i=0; i<nvar; i++) {
    a2[i]=0;
    for (j=0; j<nvar; j++) {
        cmat2[i][j]=0;
    }
}

for (; person <nused; person++) {
    p = sort2[person];
    if (strata[p] != istrat || tstop[p] < dtime) break;/*no more to add*/
    nrisk++;
    etasum += eta[p];
    ⟨fixeta⟩
    risk = exp(eta[p] - recenter) * weights[p];

    if (event[p] ==1 ){
        deaths++;
        denom2 += risk;
        meanwt += weights[p];
        newlk += weights[p]* (eta[p] - recenter);
        for (i=0; i<nvar; i++) {
            u[i] += weights[p] * covar[i][p];
            a2[i]+= risk*covar[i][p];
            for (j=0; j<=i; j++)
                cmat2[i][j] += risk*covar[i][p]*covar[j][p];
        }
    }
    else {
        denom += risk;
        for (i=0; i<nvar; i++) {
            a[i] += risk*covar[i][p];
            for (j=0; j<=i; j++)
```

```
                    cmat[i][j] += risk*covar[i][p]*covar[j][p];
                }
            }
        }
        ⟨breslow-efron⟩
    }   /* end  of accumulation loop */
```

The last step in the above loop adds terms to the loglik, score and information matrices. Assume that there were 3 tied deaths. The difference between the Efron and Breslow approximations is that for the Efron the three tied subjects are given a weight of 1/3 for the first, 2/3 for the second, and 3/3 for the third death; for the Breslow they get 3/3 for all of them. Note that `imat` is symmetric, and that the cholesky routine will utilize the upper triangle of the matrix as input, using the lower part for its own purposes. The inverse from `chinv` is also in the upper triangle.

⟨*breslow-efron*⟩=

```
 /*
 ** Add results into u and imat for all events at this time point
 */
 if (method==0 || deaths ==1) { /*Breslow */
     denom += denom2;
     newlk -= meanwt*log(denom);  /* sum of death weights*/
     for (i=0; i<nvar; i++) {
         a[i] += a2[i];
         temp = a[i]/denom;   /*mean covariate at this time */
         u[i] -= meanwt*temp;
         for (j=0; j<=i; j++) {
             cmat[i][j] += cmat2[i][j];
             imat[j][i] += meanwt*((cmat[i][j]- temp*a[j])/denom);
         }
     }
 }
 else {
     meanwt /= deaths;
     for (k=0; k<deaths; k++) {
         denom += denom2/deaths;
         newlk -= meanwt*log(denom);
         for (i=0; i<nvar; i++) {
             a[i] += a2[i]/deaths;
             temp = a[i]/denom;
             u[i] -= meanwt*temp;
             for (j=0; j<=i; j++) {
                 cmat[i][j] += cmat2[i][j]/deaths;
                 imat[j][i] += meanwt*((cmat[i][j]- temp*a[j])/denom);
             }
             }
```

```
    }
}
```

Code to process the removals:

⟨*agreg-remove*⟩=

```
/*
** subtract out the subjects whose start time is to the right
** If everyone is removed reset the totals to zero.  (This happens when
** the survSplit function is used, so it is worth checking).
*/
for (; indx1<nused; indx1++) {
    p1 = sort1[indx1];
    if (start[p1] < dtime || strata[p1] != istrat) break;
    nrisk--;
    if (nrisk ==0) {
        etasum =0;
        denom =0;
        for (i=0; i<nvar; i++) {
            a[i] =0;
            for (j=0; j<=i; j++) cmat[i][j] =0;
        }
    }
    else {
        etasum -= eta[p1];
        risk = exp(eta[p1] - recenter) * weights[p1];
        denom -= risk;
        for (i=0; i<nvar; i++) {
            a[i] -= risk*covar[i][p1];
            for (j=0; j<=i; j++)
                cmat[i][j] -= risk*covar[i][p1]*covar[j][p1];
        }
    }
}
```

The next bit of code exists for the sake of rather rare data sets. Assume that there is a time dependent covariate that rapidly climbs in such a way that the eta gets large but the range of eta stays modest. An example would be something like "payments made to date" for a portfolio of loans. Then even though the data has been centered and the global mean is fine, the current values of eta are outrageous with respect to the exp function. Since replacing eta with (eta -c) for any c does not change the likelihood, do it. Unfortunately, we can't do this once and for all: this is a step that will occur at least twice per iteration for those rare cases, e.g., eta is too small at early time points and too large at late ones.

⟨*fixeta*⟩=

```
/*
** We must avoid overflow in the exp function (~709 on Intel)
```

```
** and want to act well before that, but not take action very often.
** One of the case-cohort papers suggests an offset of -100 meaning
** that etas of 50-100 can occur in "ok" data, so make it larger
** than this.
** If the range of eta is more then log(1e16) = 37 then the data is
**  hopeless: some observations will have effectively 0 weight.  Keeping
**  the mean sensible has sufficed to keep the max in check.
*/
if (fabs(etasum/nrisk - recenter) > 200) {
    flag[1]++;  /* a count, for debugging/profiling purposes */
    temp = etasum/nrisk - recenter;
    recenter = etasum/nrisk;

    if (denom > 0) {
        /* we can skip this if there is no one at risk */
        if (fabs(temp) > 709) error("exp overflow due to covariates\n");

        temp = exp(-temp);  /* the change in scale, for all the weights */
        denom *= temp;
        for (i=0; i<nvar; i++) {
            a[i] *= temp;
            for (j=0; j<nvar; j++) {
                cmat[i][j]*= temp;
            }
        }
    }
}
```

Now, I'm finally to do the actual iteration steps. The Cox model calculation rarely gets into numerical difficulty, and when it does step halving has always been sufficient. Let $\beta^{(0)}$, $\beta^{(1)}$, etc be the iteration steps in the search for the maximum likelihood solution $\hat{\beta}$. The flow of the algorithm is

1. Iteration 0 is the loglik and etc for the intial estimates. At the end of that iteration, calculate a score test. If the user asked for 0 iterations, then don't do any singularity or infinity checks, just give them the results.

2. For the kth iteration, start with the new trial estimate $\beta^{(k)}$. This new estimate is [[beta]] in the code and the most recent successful estimate is [[oldbeta]].

3. For this new trial estimate, compute the log-likelihood, and the first and second derivatives.

4. Test if the log-likelihood if finite, has converged *and* the last estimate was not generated by step-halving. In the latter case the algorithm may *appear* to have converged but the solution is not sure. An infinite loglik is very rare, it arises when denom ¡0 due to catastrophic loss of significant digits when range(eta) is too large.

    - if converged return beta and the the other information

- if this was the last iteration, return the best beta found so far (perhaps beta, more likely oldbeta), the other information, and a warning flag.
- otherwise, compute the next guess and return to the top
  - if our latest trial guess [[beta]] made things worse use step halving: $\beta^{(k+1)}$ = oldbeta + (beta-oldbeta)/2. The assumption is that the current trial step was in the right direction, it just went too far.
  - otherwise take a Newton-Raphson step

I am particularly careful not to make a mistake that I have seen in several other Cox model programs. All the hard work is to calculate the first and second derivatives U (u) and H (imat), once we have them the next Newton-Rhapson update UH^{-1} is just a little bit more. Many programs succumb to the temptation of this "one more for free" idea, and as a consequence return $\beta^{(k+1)}$ along with the log-likelihood and variance matrix for $\beta^{(k)}$. If a user has specified for instance only 1 or 2 iterations the answers can be seriously out of joint. If iteration has gone to completion they will differ by only a gnat's eyelash, so what's the utility of the "free" update?

⟨*agfit4-iter*⟩=
```
 /* main loop */
 halving =0 ;             /* =1 when in the midst of "step halving" */
 fail =0;
 for (*iter=0; *iter<= maxiter; (*iter)++) {
     R_CheckUserInterrupt();  /* be polite -- did the user hit cntrl-C? */
     ⟨agfit4-addup⟩

     if (*iter==0) {
         loglik[0] = newlk;
         loglik[1] = newlk;
         /* compute the score test, but don't corrupt u */
         for (i=0; i<nvar; i++) a[i] = u[i];
         rank = cholesky2(imat, nvar, tol_chol);
         chsolve2(imat,nvar,a);        /* a replaced by  u *inverse(i) */
         *sctest=0;
         for (i=0; i<nvar; i++) {
            *sctest +=  u[i]*a[i];
         }
         if (maxiter==0) break;
         fail = isnan(newlk) + isinf(newlk);
         /* it almost takes malice to give a starting estimate with infinite
         **  loglik.  But if so, just give up now */
         if (fail>0) break;

         for (i=0; i<nvar; i++) {
               oldbeta[i] = beta[i];
             beta[i] += a[i];
         }
     }
```

```
else {
    fail =0;
    for (i=0; i<nvar; i++)
        if (isfinite(imat[i][i]) ==0) fail++;
    rank2 = cholesky2(imat, nvar, tol_chol);
    fail = fail + isnan(newlk) + isinf(newlk) + abs(rank-rank2);

    if (fail ==0 && halving ==0 &&
        fabs(1-(loglik[1]/newlk)) <= eps) break;  /* success! */

    if (*iter == maxiter) { /* failed to converge */
       flag[3] = 1;
       if (maxiter>1 && ((newlk -loglik[1])/ fabs(loglik[1])) < -eps) {
           /*
           ** "Once more unto the breach, dear friends, once more; ..."
           **The last iteration above was worse than one of the earlier ones,
           **  by more than roundoff error.
           ** We need to use beta and imat at the last good value, not the
           **  last attempted value. We have tossed the old imat away, so
           **  recompute it.
           ** It will happen very rarely that we run out of iterations, and
           **  even less often that it is right in the middle of halving.
           */
           for (i=0; i<nvar; i++) beta[i] = oldbeta[i];
           ⟨agfit4-addup⟩
           rank2 = cholesky2(imat, nvar, tol_chol);
           }
       break;
    }

    if (fail >0 || newlk < loglik[1]) {
        /*
        ** The routine has not made progress past the last good value.
        */
        halving++; flag[2]++;
        for (i=0; i<nvar; i++)
            beta[i] = (oldbeta[i]*halving + beta[i]) /(halving +1.0);
    }
    else {
        halving=0;
        loglik[1] = newlk;   /* best so far */
        chsolve2(imat,nvar,u);
        for (i=0; i<nvar; i++) {
            oldbeta[i] = beta[i];
            beta[i] = beta[i] +  u[i];
        }
```

```
        }
    }
} /*return for another iteration */
```

Save away the final bits, compute the inverse of imat and symmetrize it, release memory and return. If the routine did not converge (iter== maxiter), then the cholesky routine will not have been called.

⟨*agfit4-finish*⟩=

```
flag[0] = rank;
loglik[1] = newlk;
chinv2(imat, nvar);
for (i=0; i<nvar; i++) {
    beta[i] *= scale[i];  /* return to original scale */
    u[i] /= scale[i];
    imat[i][i] *= scale[i] * scale[i];
    for (j=0; j<i; j++) {
        imat[j][i] *= scale[i] * scale[j];
        imat[i][j] = imat[j][i];
    }
}
UNPROTECT(nprotect);
return(rlist);
```

2.4 Predicted survival

The `survfit` method for a Cox model produces individual survival curves. As might be expected these have much in common with ordinary survival curves, and share many of the same methods. The primary differences are first that a predicted curve always refers to a particular set of covariate values. It is often the case that a user wants multiple values at once, in which case the result will be a matrix of survival curves with a row for each time and a column for each covariate set. The second is that the computations are somewhat more difficult.

The input arguments are

formula a fitted object of class 'coxph'. The argument name of 'formula' is historic, from when the survfit function was not a generic and only did Kaplan-Meier type curves.

newdata contains the data values for which curves should be produced, one per row

se.fit TRUE/FALSE, should standard errors be computed.

individual a particular option for time-dependent covariates

stype survival type for the formula 1=direct 2= exp

ctype cumulative hazard, 1=Nelson-Aalen, 2= corrected for ties

censor if FALSE, remove any times that have no events from the output. This is for backwards compatability with older versions of the code.

id replacement and extension for the individual argument

start.time Start a curve at a later timepoint than zero.

influence whether to return the influence matrix

All the other arguments are common to all the methods, refer to the help pages.

Other survival routines have id and cluster options; this routine inherits those variables from coxph. If coxph did a robust variance, this routine will do one also.

```
⟨survfit.coxph⟩=
 survfit.coxph <-
   function(formula, newdata, se.fit=TRUE, conf.int=.95, individual=FALSE,
            stype=2, ctype,
            conf.type=c("log", "log-log", "plain", "none", "logit", "arcsin"),
            censor=TRUE, start.time, id, influence=FALSE,
            na.action=na.pass, type, ...) {

       Call <- match.call()
       Call[[1]] <- as.name("survfit")  #nicer output for the user
       object <- formula     #'formula' because it has to match survfit

       ⟨survfit.coxph-setup1⟩
       ⟨survfit.coxph-setup2⟩
       ⟨survfit.coxph-setup2b⟩
       ⟨survfit.coxph-setup2c⟩
       ⟨survfit.coxph-setup3⟩
       if (missing(newdata)) {
           if (inherits(formula, "coxphms"))
               stop ("newdata is required for multi-state models")
           risk2 <- 1
       }
       else {
           if (length(object$means))
               risk2 <- exp(c(x2 %*% beta) + offset2 - xcenter)
           else risk2 <- exp(offset2 - xcenter)
       }
       ⟨survfit.coxph-result⟩
       ⟨survfit.coxph-finish⟩
       }
```

The third line `as.name('survfit')` causes the printout to say 'survfit' instead of 'survfit.coxph'.

The setup for the has three main phases, first of course to sort out the options the user has given us, second to rebuild the data frame, X matrix, etc from the original Cox model, and third to create variables from the new data set. In the code below x2, y2, strata2, id2, etc. are

variables from the new data, X, Y, strata etc from the old. One exception to the pattern is id= argument, oldid = id from original data, id2 = id from new.

If the newdata argument is missing we use `object$means` as the default value. This choice has lots of statistical shortcomings, particularly in a stratified model, but is common in other packages and a historic option here. If stype is missing we use the standard approach of exp(cumulative hazard), and ctype is pulled from the Cox model. That is, the `coxph` computation used for `ties='breslow'` is the same as the Nelson-Aalen hazard estimate, and the Efron approximation the tie-corrected hazard.

One particular special case (that gave me fits for a while) is when there are non-heirarchical models, for example `age + age:sex`. The fit of such a model will *not* be the same using the variable `age2 <- age-50`; I originally thought it was a flaw induced by my subtraction. The routine simply cannot give a sensible curve for a model like this. The issue continued to surprise me each time I rediscovered it, leading to an error message for my own protection. I'm not convinced at this time that there is a sensible survival curve that *could* be calculated for such a model. A model with `age + age:strata(sex)` will be ok, because the coxph routine treats this last term as though it had a * in it, i.e., fits a stratified model.

⟨*survfit.coxph-setup1*⟩=
```
Terms  <- terms(object)
robust <- !is.null(object$naive.var)   # did the coxph model use robust var?

if (!is.null(attr(object$terms, "specials")$tt))
    stop("The survfit function can not process coxph models with a tt term")

if (!missing(type)) {  # old style argument
    if (!missing(stype) || !missing(ctype))
        warning("type argument ignored")
    else {
        temp1 <- c("kalbfleisch-prentice", "aalen", "efron",
                   "kaplan-meier", "breslow", "fleming-harrington",
                   "greenwood", "tsiatis", "exact")

        survtype <- match(match.arg(type, temp1), temp1)
        stype <- c(1,2,2,1,2,2,2,2,2)[survtype]
        if (stype!=1) ctype <-c(1,1,2,1,1,2,1,1,1)[survtype]
    }
}
if (missing(ctype)) {
    # Use the appropriate one from the model
    temp1 <- match(object$method, c("exact", "breslow", "efron"))
    ctype <- c(1,1,2)[temp1]
}
else if (!(ctype %in% 1:2)) stop ("ctype must be 1 or 2")
if (!(stype %in% 1:2)) stop("stype must be 1 or 2")

if (!se.fit) conf.type <- "none"
```

```
else conf.type <- match.arg(conf.type)

tfac <- attr(Terms, 'factors')
temp <- attr(Terms, 'specials')$strata
has.strata <- !is.null(temp)
if (has.strata) {
    stangle = untangle.specials(Terms, "strata")  #used multiple times, later
    # Toss out strata terms in tfac before doing the test 1 line below, as
    #  strata end up in the model with age:strat(grp) terms or *strata() terms
    #  (There might be more than one strata term)
    for (i in temp) tfac <- tfac[,tfac[i,] ==0]  # toss out strata terms
}
if (any(tfac >1))
    stop("not able to create a curve for models that contain an interaction without the lower orde

Terms <- object$terms
n <- object$n[1]
if (!has.strata) strata <- NULL
else strata <- object$strata

if (!missing(individual)) warning("the 'id' option supersedes 'individual'")
missid <- missing(id) # I need this later, and setting id below makes
                      # "missing(id)" always false

if (!missid) individual <- TRUE
else if (missid && individual) id <- rep(0L,n)  #dummy value
else id <- NULL

if (individual & missing(newdata)) {
    stop("the id option only makes sense with new data")
}
```

In two places below we need to know if there are strata by covariate interactions, which requires looking at attributes of the terms object. The factors attribute will have a row for the strata variable, or maybe more than one (multiple strata terms are legal). If it has a 1 in a column that corresponds to something of order 2 or greater, that is a strata by covariate interaction.

⟨*survfit.coxph-setup1*⟩=
```
if (has.strata) {
    temp <- attr(Terms, "specials")$strata
    factors <- attr(Terms, "factors")[temp,]
    strata.interaction <- any(t(factors)*attr(Terms, "order") >1)
}
```

I need to retrieve a copy of the original data. We always need the X matrix and y, both of which might be found in the data object. If the fit was a multistate model, the original call

included either strata, offset, weights, or id, or if either x or y are missing from the `coxph` object, then the model frame will need to be reconstructed. We have to use `object['x']` instead of `object$x` since the latter will pick off the `xlevels` component if the `x` component is missing (which is the default).

⟨*survfit.coxph-setup1*⟩=
```
 coxms <- inherits(object, "coxphms")
 if (coxms || is.null(object$y) || is.null(object[['x']]) ||
     !is.null(object$call$weights) || !is.null(object$call$id) ||
     (has.strata && is.null(object$strata)) ||
     !is.null(attr(object$terms, 'offset'))) {

     mf <- stats::model.frame(object)
     }
 else mf <- NULL  #useful for if statements later
```

For a single state model we can grab the X matrix off the model frame, for multistate some more work needs to be done. We have to repeat some lines from coxph, but to do that we need some further material. We prefer `object$y` to model.response, since the former will have been passed through aeqSurv with the options the user specified. For a multi-state model, however, we do have to recreate since the saved y has been expanded. In that case observe the saved status of timefix. Old saved objects might not have that element, if missing assume TRUE.

⟨*survfit.coxph-setup2*⟩=
```
 position <- NULL
 Y <- object[['y']]
 if (is.null(mf)) {
     weights <- object$weights  # let offsets/weights be NULL until needed
     offset <- NULL
     X <- object[['x']]
 }
 else {
     weights <- model.weights(mf)
     offset <- model.offset(mf)
     X <- model.matrix.coxph(object, data=mf)
     if (is.null(Y) || coxms) {
         Y <- model.response(mf)
         if (is.null(object$timefix) || object$timefix) Y <- aeqSurv(Y)
     }
     oldid <- model.extract(mf, "id")
     if (length(oldid) && ncol(Y)==3) position <- survflag(Y, oldid)
     else position <- NULL
     if (!coxms && (nrow(Y) != object$n[1]))
         stop("Failed to reconstruct the original data set")
     if (has.strata) {
         if (length(strata)==0) {
             if (length(stangle$vars) ==1) strata <- mf[[stangle$vars]]
```

```
            else strata <- strata(mf[, stangle$vars], shortlabel=TRUE)
        }
    }

}
```

If a model frame was created, then it is trivial to grab `y` from the new frame and compare it to `object$y` from the original one. This is to avoid nonsense results that arise when someone changes the data set under our feet. We can only check the size: with the addition of aeqSurv other packages were being flagged for tiny discrepancies. Later note: this check does not work for multi-state models, and we don't *have* to have it. Removed by using if (FALSE) so as to preserve the code for future consideration.

⟨*survfit.coxph-setup2b*⟩=

```
if (FALSE) {
if (!is.null(mf)){
    y2 <- object[['y']]
    if (!is.null(y2)) {
        if (ncol(y2) != ncol(Y) || length(y2) != length(Y))
            stop("Could not reconstruct the y vector")
    }
}
}
type <- attr(Y, 'type')
if (!type %in% c("right", "counting", "mright", "mcounting"))
    stop("Cannot handle \"", type, "\" type survival data")

if (!missing(start.time)) {
    if (!is.numeric(start.time) || length(start.time) > 1)
        stop("start.time must be a single numeric value")
    # Start the curves after start.time
    # To do so, remove any rows of the data with an endpoint before that
    #  time.
    if (ncol(Y)==3) {
        keep <- Y[,2] > start.time
        Y[keep,1] <- pmax(Y[keep,1], start.time)
    }
    else keep <- Y[,1] > start.time
    if (!any(Y[keep, ncol(Y)]==1))
        stop("start.time argument has removed all endpoints")
    Y <- Y[keep,,drop=FALSE]
    X <- X[keep,,drop=FALSE]
    if (!is.null(offset)) offset <- offset[keep]
    if (!is.null(weights)) weights <- weights[keep]
    if (!is.null(strata))  strata <- strata[keep]
    if (length(id) >0 ) id <- id[keep]
    if (length(position) >0) position <- position[keep]
```

```
    n <- nrow(Y)
 }
```

In the above code we see id twice. The first, kept as `oldid` is the identifier variable for subjects in the original data set, and is needed whenever it contained subjects with more than one row. The second is the user variable of this call, and is used to define multiple rows for a new subject. The latter usage should be rare but we need to allow for it.

If a variable is deemed redundant the `coxph` routine will have set its coefficient to NA as a marker. We want to ignore that coefficient: treating it as a zero has the desired effect. Another special case is a null model, having either 1 or only an offset on the right hand side. In that case we create a dummy covariate to allow the rest of the code to work without special if/else. The last special case is a model with a sparse frailty term. We treat the frailty coefficients as 0 variance (in essence as an offset). The frailty is removed from the model variables but kept in the risk score. This isn't statistically very defensible, but it is backwards compatatble. A non-sparse frailty does not need special code and works out like any other variable.

Center the risk scores by subtracting $\overline{x}\hat{\beta}$ from each. The reason for this is to avoid huge values when calculating $\exp(X\beta)$; this would happen if someone had a variable with a mean of 1000 and a variance of 1. Any constant can be subtracted, mathematically the results are identical as long as the same values are subtracted from the old and new X data. The mean is used because it is handy, we just need to get $X\beta$ in the neighborhood of zero.

⟨*survfit.coxph-setup2c*⟩=

```
 if (length(object$means) ==0) { # a model with only an offset term
     # Give it a dummy X so the rest of the code goes through
     #  (This case is really rare)
     # se.fit <- FALSE
     X <- matrix(0., nrow=n, ncol=1)
     if (is.null(offset)) offset <- rep(0, n)
     xcenter <- mean(offset)
     coef <- 0.0
     varmat <- matrix(0.0, 1, 1)
     risk <- rep(exp(offset- mean(offset)), length=n)
 }
 else {
     varmat <- object$var
     beta <- ifelse(is.na(object$coefficients), 0, object$coefficients)
     if (is.null(offset)) xcenter <- sum(object$means * beta)
     else xcenter <- sum(object$means * beta)+ mean(offset)
     if (!is.null(object$frail)) {
        keep <- !grepl("frailty(", dimnames(X)[[2]], fixed=TRUE)
        X <- X[,keep, drop=F]
     }

     if (is.null(offset)) risk <- c(exp(X%*% beta - xcenter))
     else     risk <- c(exp(X%*% beta + offset - xcenter))
 }
```

The `risk` vector and `x` matrix come from the original data, and are the raw data for the survival curve and its variance. We also need the risk score $\exp(X\beta)$ for the target subject(s).

- For predictions with time-dependent covariates the user will have either included an `id` statement (newer style) or specified the `individual=TRUE` option. If the latter, then `newdata` is presumed to contain only a single indivual represented by multiple rows. If the former then the `id` variable marks separate individuals. In either case we need to retrieve the covariates, strata, and repsonse from the new data set.
- For ordinary predictions only the covariates are needed.
- If newdata is not present we assume that this is the ordinary case, and use the value of `object$means` as the default covariate set. This is not ideal statistically since many users view this as an "average" survival curve, which it is not.

When grabbing [newdata] we want to use model.frame processing, both to handle missing values correctly and, perhaps more importantly, to correctly map any factor variables between the original fit and the new data. (The new data will often have only one of the original levels represented.) Also, we want to correctly handle data-dependent nonlinear terms such as ns and pspline. However, the simple call found in predict.lm, say, `model.frame(Terms, data=newdata, ..` isn't used here for a few reasons. The first is a decision on our part that the user should not have to include unused terms in the newdata: sometimes we don't need the response and sometimes we do. Second, if there are strata, the user may or may not have included strata variables in their data set and we need to act accordingly. The third is that we might have an `id` statement in this call, which is another variable to be fetched. At one time we dealt with cluster() terms in the formula, but the coxph routine has already removed those for us. Finally, note that there is no ability to use sparse frailties and newdata together; it is a hard case and so rare as to not be worth it.

First, remove unnecessary terms from the orginal model formula. If `individual` is false then the repsonse variable can go.

The dataClasses and predvars attributes, if present, have elements in the same order as the first dimension of the "factors" attribute of the terms. Subscripting the terms argument does not preserve dataClasses or predvars, however. Use the pre and post subscripting factors attribute to determine what elements of them to keep. The predvars component is a call objects with one element for each term in the formula, so y age + ns(height) would lead to a predvars of length 4, element 1 is the call itself, 2 would be y, etc. The dataClasses object is a simple list.

⟨*survfit.coxph-setup3*⟩=

```
 if (missing(newdata)) {
     # If the model has interactions, print out a long warning message.
     #  People may hate it, but I don't see another way to stamp out these
     #  bad curves without backwards-incompatability.
     # I probably should complain about factors too (but never in a strata
     #   or cluster term).
     if (any(attr(Terms, "order") > 1) )
         warning("the model contains interactions; the default curve based on columm means of the X
```

```
    if (length(object$means)) {
        mf2 <- as.list(object$means)   #create a dummy newdata
        names(mf2) <- names(object$coefficients)
        mf2 <- as.data.frame(mf2)
        x2 <- matrix(object$means, 1)
    }
    else { # nothing but an offset
        mf2 <- data.frame(X=0)
        x2 <- 0
    }
    offset2 <- 0
    found.strata <- FALSE
}
else {
    if (!is.null(object$frail))
        stop("Newdata cannot be used when a model has frailty terms")

    Terms2 <- Terms
    if (!individual)  Terms2 <- delete.response(Terms)
    ⟨survfit.coxph-newdata2⟩
}
```

For backwards compatability, I allow someone to give an ordinary vector instead of a data frame (when only one curve is required). In this case I also need to verify that the elements have a name. Then turn it into a data frame, like it should have been from the beginning. (Documentation of this ability has been suppressed, however. I'm hoping people forget it ever existed.)

⟨*survfit.coxph-newdata2*⟩=

```
if (is.vector(newdata, "numeric")) {
    if (individual) stop("newdata must be a data frame")
    if (is.null(names(newdata))) {
        stop("Newdata argument must be a data frame")
    }
    newdata <- data.frame(as.list(newdata), stringsAsFactors=FALSE)
}
```

Finally get my new model frame mf2. We allow the user to leave out any strata() variables if they so desire, *if* there are no strata by covariate interactions.

How does one check if the strata variables are or are not available in the call? My first attempt at this was to wrap the call in a try() construct and see if it failed. This doesn't work.

- What if there is no strata variable in newdata, but they do have, by bad luck, a variable of the same name in their main directory?
- It would seem like changing the environment to NULL would be wise, so that we don't find variables anywhere but in the data argument, a sort of sandboxing. Not wise: you then won't find functions like "log".

- We don't dare modify the environment of the formula at all. It is needed for the sneaky caller who uses his own function inside the formula, 'mycosine' say, and that function can only be found if we retain the environment.

One way out of this is to evaluate each of the strata terms (there can be more than one) one at a time, in an environment that knows nothing except "list" and a fake definition of "strata", and newdata. Variables that are part of the global environment won't be found. I even watch out for the case of either "strata" or "list" is the name of the stratification variable, which causes my fake strata function to return a function when said variable is not in newdata. The variable found.strata is true if ALL the strata are found, set it to false if any are missing.

⟨*survfit.coxph-newdata2*⟩=
```
 if (has.strata) {
     found.strata <- TRUE
     tempenv <- new.env(, parent=emptyenv())
     assign("strata", function(..., na.group, shortlabel, sep)
         list(...), envir=tempenv)
     assign("list", list, envir=tempenv)
     for (svar in stangle$vars) {
         temp <- try(eval(parse(text=svar), newdata, tempenv),
                     silent=TRUE)
         if (!is.list(temp) ||
             any(unlist(lapply(temp, class))== "function"))
             found.strata <- FALSE
     }

     if (!found.strata) {
         ss <- untangle.specials(Terms2, "strata")
         Terms2 <- Terms2[-ss$terms]
     }
 }

 tcall <- Call[c(1, match(c('id', "na.action"),
                               names(Call), nomatch=0))]
 tcall$data <- newdata
 tcall$formula <- Terms2
 tcall$xlev <- object$xlevels[match(attr(Terms2,'term.labels'),
                                    names(object$xlevels), nomatch=0)]
 tcall[[1L]] <- quote(stats::model.frame)
 mf2 <- eval(tcall)
```

Now, finally, extract the `x2` matrix from the just-created frame.

⟨*survfit.coxph-setup3*⟩=
```
 if (has.strata && found.strata) { #pull them off
     temp <- untangle.specials(Terms2, 'strata')
     strata2 <- strata(mf2[temp$vars], shortlabel=TRUE)
```

```
    strata2 <- factor(strata2, levels=levels(strata))
    if (any(is.na(strata2)))
        stop("New data set has strata levels not found in the original")
    # An expression like age:strata(sex) will have temp$vars= "strata(sex)"
    #  and temp$terms = integer(0).  This does not work as a subscript
    if (length(temp$terms) >0) Terms2 <- Terms2[-temp$terms]
}
else strata2 <- factor(rep(0, nrow(mf2)))

if (!robust) cluster <- NULL
if (individual) {
    if (missing(newdata))
        stop("The newdata argument must be present when individual=TRUE")
    if (!missid) {  #grab the id variable
        id2 <- model.extract(mf2, "id")
        if (is.null(id2)) stop("id=NULL is an invalid argument")
        }
    else id2 <- rep(1, nrow(mf2))

    x2 <- model.matrix(Terms2, mf2)[,-1, drop=FALSE]  #no intercept
    if (length(x2)==0) stop("Individual survival but no variables")

    offset2 <- model.offset(mf2)
    if (length(offset2) ==0) offset2 <- 0

    y2 <- model.extract(mf2, 'response')
    if (attr(y2,'type') != type)
        stop("Survival type of newdata does not match the fitted model")
    if (attr(y2, "type") != "counting")
        stop("Individual=TRUE is only valid for counting process data")
    y2 <- y2[,1:2, drop=F]  #throw away status, it's never used
}
else if (missing(newdata)) {
    if (has.strata && strata.interaction)
        stop ("Models with strata by covariate interaction terms require newdata")
    offset2 <- 0
    if (length(object$means)) {
        x2 <- matrix(object$means, nrow=1, ncol=ncol(X))
    } else {
        # model with only an offset and no new data: very rare case
        x2 <- matrix(0.0, nrow=1, ncol=1)   # make a dummy x2
    }
} else {
    offset2 <- model.offset(mf2)
    if (length(offset2) >0) offset2 <- offset2
    else offset2 <- 0
```

```
    x2 <- model.matrix(Terms2, mf2)[,-1, drop=FALSE]  #no intercept
 }
```

⟨*survfit.coxph-result*⟩=

```
 if (individual) {
     result <- coxsurv.fit(ctype, stype, se.fit, varmat, cluster,
                           Y, X, weights, risk, position, strata, oldid,
                           y2, x2, risk2, strata2, id2)
 }
 else {
     result <- coxsurv.fit(ctype, stype, se.fit, varmat, cluster,
                           Y, X, weights, risk, position, strata, oldid,
                           y2, x2, risk2)
     if (has.strata && found.strata) {
         if (is.matrix(result$surv)) {
             ⟨newstrata-fixup⟩
         }
     }
 }
```

The final bit of work. If the newdata arg contained strata then the user should not get a matrix of survival curves containing every newdata obs * strata combination, but rather a vector of curves, each one with the appropriate strata. It was faster to compute them all, however, than to use the individual=T logic. So now pick off the bits we want. The names of the curves will be the rownames of the newdata arg, if they exist.

⟨*newstrata-fixup*⟩=

```
 nr <- nrow(result$surv)  #a vector if newdata had only 1 row
 indx1 <- split(1:nr, rep(1:length(result$strata), result$strata))
 rows <- indx1[as.numeric(strata2)]  #the rows for each curve

 indx2 <- unlist(rows)  #index for time, n.risk, n.event, n.censor
 indx3 <- as.integer(strata2) #index for n and strata

 for(i in 2:length(rows)) rows[[i]] <- rows[[i]]+ (i-1)*nr #linear subscript
 indx4 <- unlist(rows)   #index for surv and std.err
 temp <- result$strata[indx3]
 names(temp) <- row.names(mf2)
 new <- list(n = result$n[indx3],
             time= result$time[indx2],
             n.risk= result$n.risk[indx2],
             n.event=result$n.event[indx2],
             n.censor=result$n.censor[indx2],
             strata = temp,
             surv= result$surv[indx4],
             cumhaz = result$cumhaz[indx4])
```

```
if (se.fit) new$std.err <- result$std.err[indx4]
result <- new
```

Finally, the last (somewhat boring) part of the code. First, if given the argument `censor=FALSE` we need to remove all the time points from the output at which there was only censoring activity. This action is mostly for backwards compatability with older releases that never returned censoring times. Second, add in the variance and the confidence intervals to the result. The code is nearly identical to that in survfitKM.

⟨*survfit.coxph-finish*⟩=

```
if (!censor) {
    kfun <- function(x, keep){ if (is.matrix(x)) x[keep,,drop=F]
                               else if (length(x)==length(keep)) x[keep]
                               else x}
    keep <- (result$n.event > 0)
    if (!is.null(result$strata)) {
        temp <- factor(rep(names(result$strata), result$strata),
                       levels=names(result$strata))
        result$strata <- c(table(temp[keep]))
        }
    result <- lapply(result, kfun, keep)
    }
result$logse = TRUE   # this will migrate further in

if (se.fit && conf.type != "none") {
    ci <- survfit_confint(result$surv, result$std.err, logse=result$logse,
                          conf.type, conf.int)
    result <- c(result, list(lower=ci$lower, upper=ci$upper,
                             conf.type=conf.type, conf.int=conf.int))
}

if (!missing(start.time)) result$start.time <- start.time

result$call <- Call
class(result) <- c('survfitcox', 'survfit')
result
```

Now, we're ready to do the main compuation. The code has gone through multiple iteration as options and complexity increased.

Computations are separate for each strata, and each strata will have a different number of time points in the result. Thus we can't preallocate a matrix. Instead we generate an empty list, one per strata, and then populate it with the survival curves. At the end we unlist the individual components one by one. This is memory efficient, the number of curves is usually small enough that the "for" loop is no great cost, and it's easier to see what's going on than C code. The computational exception is a model with thousands of strata, e.g., a matched logistic, but in that case survival curves are useless. (That won't stop some users from trying it though.)

First, compute the baseline survival curves for each strata. If the strata was a factor produce output curves in that order, otherwise in sorted order. This fitting routine was set out as a

separate function for the sake of the rms package. They want to utilize the computation, but have a diffferent process to create the x and y data.

⟨*coxsurvfit*⟩=
```
 coxsurv.fit <- function(ctype, stype, se.fit, varmat, cluster,
                         y, x, wt, risk, position, strata, oldid,
                         y2, x2, risk2, strata2, id2, unlist=TRUE) {

     if (missing(strata) || length(strata)==0) strata <- rep(0L, nrow(y))

     if (is.factor(strata)) ustrata <- levels(strata)
     else                   ustrata <- sort(unique(strata))
     nstrata <- length(ustrata)
     survlist <- vector('list', nstrata)
     names(survlist) <- ustrata
     survtype <- if (stype==1) 1 else ctype+1
     vartype <- survtype
     if (is.null(wt)) wt <- rep(1.0, nrow(y))
     if (is.null(strata)) strata <- rep(1L, nrow(y))
     for (i in 1:nstrata) {
         indx <- which(strata== ustrata[i])
         survlist[[i]] <- agsurv(y[indx,,drop=F], x[indx,,drop=F],
                                 wt[indx], risk[indx],
                                 survtype, vartype)
         }
     ⟨survfit.coxph-compute⟩

     if (unlist) {
         if (length(result)==1) { # the no strata case
             if (se.fit)
                 result[[1]][c("n", "time", "n.risk", "n.event", "n.censor",
                           "surv", "cumhaz", "std.err")]
             else result[[1]][c("n", "time", "n.risk", "n.event", "n.censor",
                           "surv", "cumhaz")]
         }
         else {
             ⟨survfit.coxph-unlist⟩
         }
     }
     else {
         names(result) <- ustrata
         result
     }
 }
```

In an ordinary survival curve object with multiple strata, as produced by `survfitKM`, the time, survival and etc components are each a single vector that contains the results for strata

1, followed by strata 2, ... The strata compontent is a vector of integers, one per strata, that gives the number of elements belonging to each stratum. The reason is that each strata will have a different number of observations, so that a matrix form was not viable, and the underlying C routines were not capable of handling lists (the code predates the .Call function by a decade). The underlying computation of `survfitcoxph.fit` naturally creates the list form, we unlist it to `survfit` form as our last action unless the caller requests otherwise.

⟨*survfit.coxph-unlist*⟩=
```
 temp <-list(n   =    unlist(lapply(result, function(x) x$n),
                            use.names=FALSE),
             time=    unlist(lapply(result, function(x) x$time),
                            use.names=FALSE),
             n.risk=  unlist(lapply(result, function(x) x$n.risk),
                            use.names=FALSE),
             n.event= unlist(lapply(result, function(x) x$n.event),
                            use.names=FALSE),
             n.censor=unlist(lapply(result, function(x) x$n.censor),
                            use.names=FALSE),
             strata = sapply(result, function(x) length(x$time)))
 names(temp$strata) <- names(result)

 if ((missing(id2) || is.null(id2)) && nrow(x2)>1) {
      temp$surv <- t(matrix(unlist(lapply(result,
                        function(x) t(x$surv)), use.names=FALSE),
                            nrow= nrow(x2)))
      dimnames(temp$surv) <- list(NULL, row.names(x2))
      temp$cumhaz <- t(matrix(unlist(lapply(result,
                        function(x) t(x$cumhaz)), use.names=FALSE),
                            nrow= nrow(x2)))
      if (se.fit)
          temp$std.err <- t(matrix(unlist(lapply(result,
                           function(x) t(x$std.err)), use.names=FALSE),
                                    nrow= nrow(x2)))
      }
 else {
     temp$surv <- unlist(lapply(result, function(x) x$surv),
                         use.names=FALSE)
     temp$cumhaz <- unlist(lapply(result, function(x) x$cumhaz),
                         use.names=FALSE)
     if (se.fit)
         temp$std.err <- unlist(lapply(result,
                        function(x) x$std.err), use.names=FALSE)
     }
 temp
```

For `individual=FALSE` we have a second dimension, namely each of the target covariate sets (if there are multiples). Each of these generates a unique set of survival and variance(survival)

values, but all of the same size since each uses all the strata. The final output structure in this case has single vectors for the time, number of events, number censored, and number at risk values since they are common to all the curves, and a matrix of survival and variance estimates, one column for each of the distinct target values. If Λ_0 is the baseline cumulative hazard from the above calculation, then $r_i\Lambda_0$ is the cumulative hazard for the ith new risk score r_i. The variance has two parts, the first of which is $r_i^2 H_1$ where H_1 is returned from the `agsurv` routine, and the second is

$$H_2(t) = d'(t) V d(t)$$
$$d(t) = \int_0^t [z - \overline{x}(s)] d\Lambda(s)$$

V is the variance matrix for β from the fitted Cox model, and $d(t)$ is the distance between the target covariate z and the mean of the original data, summed up over the interval from 0 to t. Essentially the variance in $\hat{\beta}$ has a larger influence when prediction is far from the mean. The function below takes the basic curve from the list and multiplies it out to matrix form.

⟨*survfit.coxph-compute*⟩=
```
 expand <- function(fit, x2, varmat, se.fit) {
     if (survtype==1)
         surv <- cumprod(fit$surv)
     else surv <- exp(-fit$cumhaz)

     if (is.matrix(x2) && nrow(x2) >1) {  #more than 1 row in newdata
         fit$surv <- outer(surv, risk2, '^')
         dimnames(fit$surv) <- list(NULL, row.names(x2))
         if (se.fit) {
             varh <- matrix(0., nrow=length(fit$varhaz), ncol=nrow(x2))
             for (i in 1:nrow(x2)) {
                 dt <- outer(fit$cumhaz, x2[i,], '*') - fit$xbar
                 varh[,i] <- (cumsum(fit$varhaz) + rowSums((dt %*% varmat)* dt))*
                     risk2[i]^2
                 }
             fit$std.err <- sqrt(varh)
             }
         fit$cumhaz <- outer(fit$cumhaz, risk2, '*')
         }
     else {
         fit$surv <- surv^risk2
         if (se.fit) {
             dt <-  outer(fit$cumhaz, c(x2)) - fit$xbar
             varh <- (cumsum(fit$varhaz) + rowSums((dt %*% varmat)* dt)) *
                 risk2^2
             fit$std.err <- sqrt(varh)
             }
         fit$cumhaz <- fit$cumhaz * risk2
```

```
    }
fit
}
```

In the lines just above: I have a matrix `dt` with one row per death time and one column per variable. For each row d_i separately we want the quadratic form $d_i V d_i'$. The first matrix product can be done for all rows at once: found in the inner parenthesis. Ordinary (not matrix) multiplication followed by rowsums does the rest in one fell swoop.

Now, if `id2` is missing we can simply apply the `expand` function to each strata. For the case with `id2` not missing, we create a single survival curve for each unique id (subject). A subject will spend blocks of time with different covariate sets, sometimes even jumping between strata. Retrieve each one and save it into a list, and then sew them together end to end. The `n` component is the number of observations in the strata — but this subject might visit several. We report the first one they were in for printout. The `time` component will be cumulative on this subject's scale. Counting this is a bit trickier than I first thought. Say that the subject's first interval goes from 1 to 10, with observed time points in that interval at 2, 5, and 7, and a second interval from 12 to 20 with observed time points in the data of 15 and 18. On the subject's time scale things happen at days 1, 4, 6, 12 and 15. The deltas saved below are 2-1, 5-2, 7-5, 3+ 14-12, 17-14. Note the 3+ part, kept in the `timeforward` variable. Why all this "adding up" nuisance? If the subject spent time in two strata, the second one might be on an internal time scale of 'time since entering the strata'. The two intervals in newdata could be 0–10 followed by 0–20. Time for the subject can't go backwards though: the change between internal/external time scales is a bit like following someone who was stepping back and forth over the international date line.

In the code the `indx` variable points to the set of times that the subject was present, for this row of the new data. Note the $>$ on one end and $\leq$ on the other. If someone's interval 1 was 0–10 and interval 2 was 10–20, and there happened to be a jump in the baseline survival curve at exactly time 10 (someone else died), that jump is counted only in the first interval.

⟨*survfit.coxph-compute*⟩=

```
 if (missing(id2) || is.null(id2))
     result <- lapply(survlist, expand, x2, varmat, se.fit)
 else {
     onecurve <- function(slist, x2, y2, strata2,  risk2, se.fit) {
         ntarget <- nrow(x2)  #number of different time intervals
         surv <- vector('list', ntarget)
         n.event <- n.risk <- n.censor <- varh1 <- varh2 <-  time <- surv
         hazard  <- vector('list', ntarget)
         stemp <- as.integer(strata2)
         timeforward <- 0
         for (i in 1:ntarget) {
             slist <- survlist[[stemp[i]]]
             indx <- which(slist$time > y2[i,1] & slist$time <= y2[i,2])
             if (length(indx)==0) {
                 timeforward <- timeforward + y2[i,2] - y2[i,1]
                 # No deaths or censors in user interval.  Possible
                 # user error, but not uncommon at the tail of the curve.
```

```
        }
        else {
            time[[i]] <- diff(c(y2[i,1], slist$time[indx])) #time increments
            time[[i]][1] <- time[[i]][1] + timeforward
            timeforward <- y2[i,2] - max(slist$time[indx])

            hazard[[i]] <- slist$hazard[indx]*risk2[i]
            if (survtype==1) surv[[i]] <- slist$surv[indx]^risk2[i]

            n.event[[i]] <- slist$n.event[indx]
            n.risk[[i]]  <- slist$n.risk[indx]
            n.censor[[i]]<- slist$n.censor[indx]
            dt <-  outer(slist$cumhaz[indx], x2[i,]) - slist$xbar[indx,,drop=F]
            varh1[[i]] <- slist$varhaz[indx] *risk2[i]^2
            varh2[[i]] <- rowSums((dt %*% varmat)* dt) * risk2[i]^2
        }
    }

    cumhaz <- cumsum(unlist(hazard))
    if (survtype==1) surv <- cumprod(unlist(surv))  #increments (K-M)
    else surv <- exp(-cumhaz)

    if (se.fit)
        list(n=as.vector(table(strata)[stemp[1]]),
               time=cumsum(unlist(time)),
               n.risk = unlist(n.risk),
               n.event= unlist(n.event),
               n.censor= unlist(n.censor),
               surv = surv,
               cumhaz= cumhaz,
               std.err = sqrt(cumsum(unlist(varh1)) + unlist(varh2)))
    else list(n=as.vector(table(strata)[stemp[1]]),
               time=cumsum(unlist(time)),
               n.risk = unlist(n.risk),
               n.event= unlist(n.event),
               n.censor= unlist(n.censor),
               surv = surv,
               cumhaz= cumhaz)
}

if (all(id2 ==id2[1])) {
    result <- list(onecurve(survlist, x2, y2, strata2, risk2, se.fit))
}
else {
    uid <- unique(id2)
    result <- vector('list', length=length(uid))
```

```
        for (i in 1:length(uid)) {
            indx <- which(id2==uid[i])
            result[[i]] <- onecurve(survlist, x2[indx,,drop=FALSE],
                                    y2[indx,,drop=FALSE],
                                    strata2[indx],  risk2[indx], se.fit)
        }
        names(result) <- uid
    }
}
```

Next is the code for the `agsurv` function, which actually does the work. The estimates of survival are the Kalbfleisch-Prentice (KP), Breslow, and Efron. Each has an increment at each unique death time. First a bit of notation: $Y_i(t)$ is 1 if bservation i is "at risk" at time t and 0 otherwise. For a simple surivival (`ncol(y)==2`) a subject is at risk until the time of censoring or death (first column of `y`). For (start, stop] data (`ncol(y)==3`) a subject becomes a part of the risk set at start+0 and stays through stop. $dN_i(t)$ will be 1 if subject i had an event at time t. The risk score for each subject is $r_i = \exp(X_i\beta)$.

The Breslow increment at time t is $\sum w_i dN_i(t) / \sum w_i r_i Y_i(t)$, the number of events at time t over the number at risk at time t. The final survival is `exp(-cumsum(increment))`.

The Kalbfleish-Prentice increment is a multiplicative term z which is the solution to the equation

$$\sum w_i r_i Y_i(t) = \sum dN_i(t) w_i \frac{r_i}{1 - z(t)^{r_i}}$$

The left hand side is the weighted number at risk at time t, the right hand side is a sum over the tied events at that time. If there is only one event the equation has a closed form solution. If not, and knowing the solution must lie between 0 and 1, we do 35 steps of bisection to get a solution within 1e-8. An alternative is to use the -log of the Breslow estimate as a starting estimate, which is faster but requires a more sophisticated iteration logic. The final curve is $\prod_t z(t)^{r_c}$ where r_c is the risk score for the target subject.

The Efron estimate can be viewed as a modified Breslow estimate under the assumption that tied deaths are not really tied – we just don't know the order. So if there are 3 subjects who die at some time t we will have three psuedo-terms for t, $t + \epsilon$, and $t + 2\epsilon$. All 3 subjects are present for the denominator of the first term, 2/3 of each for the second, and 1/3 for the third terms denominator. All contribute 1/3 of the weight to each numerator (1/3 chance they were the one to die there). The formulas will require $\sum w_i dN_i(t)$, $\sum w_i r_i dN_i(t)$, and $\sum w_i X_i dN_i(t)$, i.e., the sums only over the deaths.

For simple survival data the risk sum $\sum w_i r_i Y_i(t)$ for all the unique death times t is fast to compute as a cumulative sum, starting at the longest followup time and summing towards the shortest. There are two algorithms for (start, stop] data.

- Do a separate sum at each death time. The problem is for very large data sets. For each death time the selection `(start<t & stop>=t)` is $O(n)$ and can take more time then all the remaining calculations together.

- Use the difference of two cumulative sums, one ordered by start time and one ordered by stop time. This is $O(2n)$ for the intial sums. The problem here is potential round off

error if the sums get large. This issue is mostly precluded by subtracting means first, and avoiding intervals that don't overlap an event time.

We compute the extended number still at risk — all whose stop time is $\geq$ each unique death time — in the vector `xin`. From this we have to subtract all those who haven't actually entered yet found in `xout`. Remember that (3,20] enters at time 3+. The total at risk at any time is the difference between them. Output is only for the stop times; a call to approx is used to reconcile the two time sets. The `irisk` vector is for the printout, it is a sum of weighted counts rather than weighted risk scores.

⟨*agsurv*⟩=
```
 agsurv <- function(y, x, wt, risk, survtype, vartype) {
     nvar <- ncol(as.matrix(x))
     status <- y[,ncol(y)]
     dtime <- y[,ncol(y) -1]
     death <- (status==1)

     time <- sort(unique(dtime))
     nevent <- as.vector(rowsum(wt*death, dtime))
     ncens  <- as.vector(rowsum(wt*(!death), dtime))
     wrisk <- wt*risk
     rcumsum <- function(x) rev(cumsum(rev(x))) # sum from last to first
     nrisk <- rcumsum(rowsum(wrisk, dtime))
     irisk <- rcumsum(rowsum(wt, dtime))
     if (ncol(y) ==2) {
         temp2  <- rowsum(wrisk*x, dtime)
         xsum   <- apply(temp2, 2, rcumsum)
         }
     else {
         delta <- min(diff(time))/2
         etime <- c(sort(unique(y[,1])), max(y[,1])+delta)  #unique entry times
         indx  <- approx(etime, 1:length(etime), time, method='constant',
                         rule=2, f=1)$y
         esum <- rcumsum(rowsum(wrisk, y[,1]))  #not yet entered
         nrisk <- nrisk - c(esum,0)[indx]
         irisk <- irisk - c(rcumsum(rowsum(wt, y[,1])),0)[indx]
         xout   <- apply(rowsum(wrisk*x, y[,1]), 2, rcumsum) #not yet entered
         xin  <- apply(rowsum(wrisk*x, dtime), 2, rcumsum) # dtime or alive
         xsum  <- xin - (rbind(xout,0))[indx,,drop=F]
         }

     ndeath <- rowsum(status, dtime)  #unweighted death count
```

The KP estimate requires a short C routine to do the iteration efficiently, and the Efron estimate needs a second C routine to efficiently compute the partial sums.

⟨*agsurv*⟩=
```
     ntime  <- length(time)
```

```
if (survtype ==1) {  #Kalbfleisch-Prentice
    indx <- (which(status==1))[order(dtime[status==1])] #deaths
    km <- .C(Cagsurv4,
         as.integer(ndeath),
         as.double(risk[indx]),
         as.double(wt[indx]),
         as.integer(ntime),
         as.double(nrisk),
         inc = double(ntime))
}

if (survtype==3 || vartype==3) {  # Efron approx
    xsum2 <- rowsum((wrisk*death) *x, dtime)
    erisk <- rowsum(wrisk*death, dtime)  #risk score sums at each death
    tsum  <- .C(Cagsurv5,
                as.integer(length(nevent)),
                as.integer(nvar),
                as.integer(ndeath),
                as.double(nrisk),
                as.double(erisk),
                as.double(xsum),
                as.double(xsum2),
                sum1 = double(length(nevent)),
                sum2 = double(length(nevent)),
                xbar = matrix(0., length(nevent), nvar))
}
haz <- switch(survtype,
                nevent/nrisk,
                nevent/nrisk,
                nevent* tsum$sum1)
varhaz <- switch(vartype,
                nevent/(nrisk *
                          ifelse(nevent>=nrisk, nrisk, nrisk-nevent)),
                nevent/nrisk^2,
                nevent* tsum$sum2)
xbar <- switch(vartype,
               (xsum/nrisk)*haz,
               (xsum/nrisk)*haz,
               nevent * tsum$xbar)

result <- list(n= nrow(y), time=time, n.event=nevent, n.risk=irisk,
               n.censor=ncens, hazard=haz,
               cumhaz=cumsum(haz), varhaz=varhaz, ndeath=ndeath,
               xbar=apply(matrix(xbar, ncol=nvar),2, cumsum))
if (survtype==1) result$surv <- km$inc
result
```

```
}
```

The arguments to this function are the number of unique times n, which is the length of the vectors ndeath (number at each time), denom, and the returned vector km. The risk and wt vectors contain individual values for the subjects with an event. Their length will be equal to sum(ndeath).

⟨*agsurv4*⟩=

```
#include "survS.h"
#include "survproto.h"

void agsurv4(Sint   *ndeath,   double *risk,    double *wt,
             Sint   *sn,        double *denom,   double *km)
{
    int i,j,k, l;
    int n;  /* number of unique death times */
    double sumt, guess, inc;

    n = *sn;
    j =0;
    for (i=0; i<n; i++) {
        if (ndeath[i] ==0) km[i] =1;
        else if (ndeath[i] ==1) { /* not a tied death */
            km[i] = pow(1- wt[j]*risk[j]/denom[i], 1/risk[j]);
            }
        else { /* biscection solution */
            guess = .5;
            inc = .25;
            for (l=0; l<35; l++) { /* bisect it to death */
                sumt =0;
                for (k=j; k<(j+ndeath[i]); k++) {
                    sumt +=  wt[k]*risk[k]/(1-pow(guess, risk[k]));
                }
            if (sumt < denom[i])  guess += inc;
            else          guess -= inc;
            inc = inc/2;
            }
            km[i] = guess;
        }
        j += ndeath[i];
    }
}
```

Do a computation which is slow in R, needed for the Efron approximation. Input arguments are

n number of observations (unique death times)

d number of deaths at that time

nvar number of covariates

x1 weighted number at risk at the time

x2 sum of weights for the deaths

xsum matrix containing the cumulative sum of x values

xsum2 matrix of sums, only for the deaths

On output the values are

- d=0: the outputs are unchanged (they initialize at 0)
- d=1

  sum1 `1/x1`

  sum2 `1/x1`$\hat{2}$

  xbar `xsum/x1`$\hat{2}$
- d=2

  sum1 (1/2) `( 1/x1 + 1/(x1 - x2/2))`

  sum2 (1/2) (same terms, squared)

  xbar (1/2) `(xsum/x1`$\hat{2}$ `+ (xsum - 1/2 x3)/(x1- x2/2)`$\hat{2}$`)`
- d=3

  sum1 (1/3) `(1/x1 + 1/(x1 - x2/3) + 1/(x1 - 2*x2/3))`

  sum2 (1/3) (same terms, squared)

  xbar (1/3) `xsum/x1`$\hat{2}$ `+ (xsum - 1/3 xsum2)/(x1- x2/3)`$\hat{2}$ `+`
  `(xsum - 2/3 xsum2)/(x1- 2/3 x3)`$\hat{2}$`)`
- etc

Sum1 will be the increment to the hazard, sum2 the increment to the first term of the variance, and xbar the increment in the hazard times the mean of x at this point.

⟨*agsurv5*⟩=
```
#include "survS.h"
void agsurv5(Sint *n2,     Sint *nvar2,  Sint *dd, double *x1,
             double *x2,   double *xsum, double *xsum2,
             double *sum1, double *sum2, double *xbar) {
    double temp;
    int i,j, k, kk;
    double d;
    int n, nvar;
```

```
    n = n2[0];
    nvar = nvar2[0];

    for (i=0; i< n; i++) {
        d = dd[i];
        if (d==1){
            temp = 1/x1[i];
            sum1[i] = temp;
            sum2[i] = temp*temp;
            for (k=0; k< nvar; k++)
                xbar[i+ n*k] = xsum[i + n*k] * temp*temp;
            }
        else {
            temp = 1/x1[i];
            for (j=0; j<d; j++) {
                temp = 1/(x1[i] - x2[i]*j/d);
                sum1[i] += temp/d;
                sum2[i] += temp*temp/d;
                for (k=0; k< nvar; k++){
                    kk = i + n*k;
                    xbar[kk] += ((xsum[kk] - xsum2[kk]*j/d) * temp*temp)/d;
                    }
                }
            }
        }
    }
```

2.4.1 Multi-state models

Survival curves after a multi-state Cox model are more challenging, particularly the variance.

⟨*survfit.coxphms*⟩=

```
 survfit.coxphms <-
 function(formula, newdata, se.fit=TRUE, conf.int=.95, individual=FALSE,
          stype=2, ctype,
          conf.type=c("log", "log-log", "plain", "none", "logit", "arcsin"),
          censor=TRUE, start.time, id, influence=FALSE,
          na.action=na.pass, type, p0=NULL, ...) {

    Call <- match.call()
    Call[[1]] <- as.name("survfit")  #nicer output for the user
    object <- formula     #'formula' because it has to match survfit
    se.fit <- FALSE   #still to do
    if (missing(newdata))
        stop("multi-state survival requires a newdata argument")
```

```
if (!missing(id))
    stop("using a covariate path is not supported for multi-state")
temp <- object$stratum_map["(Baseline)",]
baselinecoef <- rbind(temp, coef= 1.0)
if (any(duplicated(temp))) {
    # We have shared hazards
    # Find rows of cmap with "ph(a:b)" type labels to find out which
    #  ones have proportionality
    rname <- rownames(object$cmap)
    phbase <- grepl("ph(", rname, fixed=TRUE)
    for (i in which(phbase)) {
        ctemp <- object$cmap[i,]
        index <- which(ctemp >0)
        baselinecoef[2, index] <- exp(object$coef[ctemp[index]])
    }
} else phbase <- rep(FALSE, nrow(object$cmap))

# process options, set up Y and the model frame, deal with start.time
⟨survfit.coxph-setup1⟩
⟨survfit.coxph-setup2⟩
istate <- model.extract(mf, "istate")
if (!missing(start.time)) {
    if (!is.numeric(start.time) || length(start.time) !=1
        || !is.finite(start.time))
        stop("start.time must be a single numeric value")
    toss <- which(Y[,ncol(Y)-1] <= start.time)
    if (length(toss)) {
        n <- nrow(Y)
        if (length(toss)==n) stop("start.time has removed all observations")
        Y <- Y[-toss,,drop=FALSE]
        X <- X[-toss,,drop=FALSE]
        weights <- weights[-toss]
        oldid <- oldid[-toss]
        istate <- istate[-toss]
    }
}

# expansion of the X matrix with stacker, set up shared hazards
⟨survfit.coxphms-setupa⟩

# risk scores, mf2, and x2
⟨survfit.coxph-setup2c⟩
⟨survfit.coxph-setup3⟩

⟨survfit.coxphms-setup3b⟩
⟨survfit.coxphms-result⟩
```

```
    cifit$call <- Call
    class(cifit) <- c("survfitms", "survfit")
    cifit
}
```

The third line `as.name('survfit')` causes the printout to say 'survfit' instead of 'survfit.coxph'.

Notice that setup is almost completely shared with survival for single state models. The major change is that we use survfitCI (non-Cox) to do all the legwork wrt the tabulation values (number at risk, etc.), while for the computation proper it is easier to make use of the same expanded data set that coxph used for a multi-state fit.

⟨*survfit.coxphms-setupa*⟩=

```
# Rebuild istate using the survcheck routine
mcheck <- survcheck2(Y, oldid, istate)
transitions <- mcheck$transitions
if (is.null(istate)) istate <- mcheck$istate
if (!identical(object$states, mcheck$states))
    stop("failed to rebuild the data set")

# Let the survfitCI routine do the work of creating the
#  overall counts (n.risk, etc).  The rest of this code then
#  replaces the surv and hazard components.
if (missing(start.time)) start.time <- min(Y[,2], 0)
# If the data has absorbing states (ones with no transitions out), then
#  remove those rows first since they won't be in the final output.
t2 <- transitions[, is.na(match(colnames(transitions), "(censored)")), drop=FALSE]
absorb <- row.names(t2)[rowSums(t2)==0]

if (is.null(weights)) weights <- rep(1.0, nrow(Y))
if (is.null(strata))  tempstrat <- rep(1L, nrow(Y))
else                  tempstrat <- strata

if (length(absorb)) droprow <- istate %in% absorb  else droprow <- FALSE

# Let survfitCI fill in the n, number at risk, number of events, etc. portions
# We will replace the pstate and cumhaz estimate with correct ones.
if (any(droprow)) {
    j <- which(!droprow)
    cifit <- survfitCI(as.factor(tempstrat[j]), Y[j,], weights[j],
                       id =oldid[j], istate= istate[j],
                       se.fit=FALSE, start.time=start.time, p0=p0)
    }
else cifit <- survfitCI(as.factor(tempstrat), Y, weights,
                        id= oldid, istate = istate, se.fit=FALSE,
                        start.time=start.time, p0=p0)
```

```
# For computing the  actual estimates it is easier to work with an
#  expanded data set.
# Replicate actions found in the coxph-multi-X chunk,
cluster <- model.extract(mf, "cluster")
xstack <- stacker(object$cmap, object$stratum_map, as.integer(istate), X, Y,
                  as.integer(strata),
                  states= object$states)
if (length(position) >0)
    position <- position[xstack$rindex]   # id was required by coxph
X <- xstack$X
Y <- xstack$Y
strata <- strata[xstack$rindex]  # strat in the model, other than transitions
transition <- xstack$transition
istrat <- xstack$strata
if (length(offset)) offset <- offset[xstack$rindex]
if (length(weights)) weights <- weights[xstack$rindex]
if (length(cluster)) cluster <- cluster[xstack$rindex]
oldid <- oldid[xstack$rindex]
if (robust & length(cluster)==0) cluster <- oldid
```

The survfit.coxph-setup3 chunk, shared with single state Cox models, has created an mf2 model frame and an x2 matrix. For multi-state, we ignore any strata variables in mf2. Create a matrix of risk scores, number of subjects by number of transitions. Different transitions often have different coefficients, so there is a risk score vector per transition.

⟨*survfit.coxphms-setup3b*⟩=

```
if (has.strata && !is.null(mf2[[stangle$vars]])){
    mf2 <- mf2[is.na(match(names(mf2), stangle$vars))]
    mf2 <- unique(mf2)
    x2 <- unique(x2)
}
temp <- coef(object, matrix=TRUE)[!phbase,,drop=FALSE] # ignore missing coefs
risk2 <- exp(x2 %*% ifelse(is.na(temp), 0, temp) - xcenter)
```

At this point we have several parts to keep straight. The data set has been expanded into a new X and Y.

- `strata` contains any strata that were specified by the user in the original fit. We do completely separate computations for each stratum: the time scale starts over, nrisk, etc. Each has a separate call to the multihaz function.
- `transtion` contains the transition to which each observation applies
- `istrat` comes from the xstack routine, and marks each strata * basline hazard combination.
- `baselinecoef` maps from baseline hazards to transitions. It has one column per transition, which hazard it points to, and a multiplier. Most multipliers will be 1.

- `hfill` is constructed below. It contains the row/column to which each column of baselinecoef is mapped, within the H matrix used to compute P(state).

The coxph routine fits all strata and transitions at once, since the loglik is a sum over strata. This routine does each stratum separately.

⟨*survfit.coxphms-result*⟩=
```
 # make the expansion map.
 # The H matrices we will need are nstate by nstate, at each time, with
 # elements that are non-zero only for observed transtions.
 states <- object$states
 nstate <- length(states)
 notcens <- (colnames(object$transitions) != "(censored)")
 trmat <- object$transitions[, notcens, drop=FALSE]
 from <- row(trmat)[trmat>0]
 from <- match(rownames(trmat), states)[from]  # actual row of H
 to   <- col(trmat)[trmat>0]
 to   <- match(colnames(trmat), states)[to]    # actual col of H
 hfill <- cbind(from, to)

 if (individual) {
     stop("time dependent survival curves are not supported for multistate")
 }
 ny <- ncol(Y)
 if (is.null(strata)) {
     fit <- multihaz(Y, X, position, weights, risk, istrat, ctype, stype,
                     baselinecoef, hfill, x2, risk2, varmat, nstate, se.fit,
                     cifit$p0, cifit$time)
     cifit$pstate <- fit$pstate
     cifit$cumhaz <- fit$cumhaz
 }
 else {
     if (is.factor(strata)) ustrata <- levels(strata)
     else                   ustrata <- sort(unique(strata))
     nstrata <- length(cifit$strata)
     itemp <- rep(1:nstrata, cifit$strata)
     timelist <- split(cifit$time, itemp)
     ustrata <- names(cifit$strata)
     tfit <- vector("list", nstrata)
     for (i in 1:nstrata) {
         indx <- which(strata== ustrata[i])  # divides the data
         tfit[[i]] <- multihaz(Y[indx,,drop=F], X[indx,,drop=F],
                               position[indx], weights[indx], risk[indx],
                               istrat[indx], ctype, stype, baselinecoef, hfill,
                               x2, risk2, varmat, nstate, se.fit,
                               cifit$p0[i,], timelist[[i]])
     }
```

```
    # do.call(rbind) doesn't work for arrays, it loses a dimension
    ntime <- length(cifit$time)
    cifit$pstate <- array(0., dim=c(ntime, dim(tfit[[1]]$pstate)[2:3]))
    cifit$cumhaz <- array(0., dim=c(ntime, dim(tfit[[1]]$cumhaz)[2:3]))
    rtemp <- split(seq(along=cifit$time), itemp)
    for (i in 1:nstrata) {
        cifit$pstate[rtemp[[i]],,] <- tfit[[i]]$pstate
        cifit$cumhaz[rtemp[[i]],,] <- tfit[[i]]$cumhaz
    }
}
cifit$newdata <- mf2
```

Finally, a routine that does all the actual work.

- The first 5 variables are for the data set that the Cox model was built on: y, x, position, risk score, istrat. Position is a flag for each obs. Is it the first of a connected string such as (10, 12) (12,19) (19,21), the last of such a string, both, or neither. 1*first + 2*last. This affects whether an obs is labeled as censored or not, nothing else.
- x2 and risk2 are the covariates and risk scores for the predicted values. These do not involve any ph(a:b) coefficients.
- baselinecoef and hfill control mapping from fittes hazards to transitions and probabilities
- p0 will be NULL if the user did not specifiy it.
- vmat is only needed for standard errors
- utime is the set of time points desired

⟨*survfit.coxphms*⟩=

```
# Compute the hazard  and survival functions
multihaz <- function(y, x, position, weight, risk, istrat, ctype, stype,
                     bcoef, hfill, x2, risk2, vmat, nstate, se.fit, p0, utime) {
    if (ncol(y) ==2) {
       sort1 <- seq.int(0, nrow(y)-1L)   # sort order for a constant
       y <- cbind(-1.0, y)               # add a start.time column, -1 in case
                                         #  there is an event at time 0
    }
    else sort1 <- order(istrat, y[,1]) -1L
    sort2 <- order(istrat, y[,2]) -1L
    ntime <- length(utime)

    # this returns all of the counts we might desire.
    storage.mode(weight) <- "double"  #failsafe
    # for Surv(time, status), position is 2 (last) for all obs
    if (length(position)==0) position <- rep(2L, nrow(y))
```

```
    fit <- .Call(Ccoxsurv2, utime, y, weight, sort1, sort2, position,
                         istrat, x, risk)
    cn <- fit$count  # 1-3 = at risk, 4-6 = events, 7-8 = censored events
                    # 9-10 = censored, 11-12 = Efron, 13-15 = entry

    if (ctype ==1) {
        denom1 <- ifelse(cn[,4]==0, 1, cn[,3])
        denom2 <- ifelse(cn[,4]==0, 1, cn[,3]^2)
    } else {
        denom1 <- ifelse(cn[,4]==0, 1, cn[,11])
        denom2 <- ifelse(cn[,4]==0, 1, cn[,12])
    }

    temp <- matrix(cn[,5] / denom1, ncol = fit$ntrans)
    hazard <- temp[,bcoef[1,]] * rep(bcoef[2,], each=nrow(temp))
    if (se.fit) {
        temp <- matrix(cn[,5] / denom2, ncol = fit$ntrans)
        varhaz <- temp[,bcoef[1,]] * rep(bcoef[2,]^2, each=nrow(temp))
    }

    # Expand the result, one "hazard set" for each row of x2
    nx2 <- nrow(x2)
    h2 <- array(0, dim=c(nrow(hazard), nx2, ncol(hazard)))
    if (se.fit) v2 <- h2
    S <- double(nstate)  # survival at the current time
    S2 <- array(0, dim=c(nrow(hazard), nx2, nstate))

    H <- matrix(0, nstate, nstate)
    if (stype==2) {
        H[hfill] <- colMeans(hazard)
        diag(H) <- diag(H) -rowSums(H)
        esetup <- survexpmsetup(H)
    }

    for (i in 1:nx2) {
        h2[,i,] <- apply(hazard %*% diag(risk2[i,]), 2, cumsum)
        if (se.fit) {
            d1 <- fit$xbar - rep(x[i,], each=nrow(fit$xbar))
            d2 <- apply(d1*hazard, 2, cumsum)
            d3 <- rowSums((d2%*% vmat) * d2)
#            v2[jj,] <- (apply(varhaz[jj,],2, cumsum) + d3) * (risk2[i])^2
        }

        S <- p0
        for (j in 1:ntime) {
```

```
            H[,] <- 0.0
            H[hfill] <- hazard[j,] *risk2[i,]
            if (stype==1) {
                diag(H) <- pmax(0, 1.0 - rowSums(H))
                S <- as.vector(S %*% H)  # don't keep any names
            }
            else {
                diag(H) <- 0.0 - rowSums(H)
                #S <- as.vector(S %*% expm(H))  # dgeMatrix issue
                S <- as.vector(S %*% survexpm(H, 1, esetup))
            }
            S2[j,i,] <- S
        }
    }
    rval <- list(time=utime, xgrp=rep(1:nx2, each=nrow(hazard)),
                 pstate=S2, cumhaz=h2)
    if (se.fit) rval$varhaz <- v2
    rval
}
```

3 The Fine-Gray model

For competing risks with ending states 1, 2, ... k, the Fine-Gray approach turns these into a set of simple 2-state Cox models:

- (not yet in state 1) $\longrightarrow$ state 1
- (not yet in state 2) $\longrightarrow$ state 2
- ...

Each of these is now a simple Cox model, assuming that we are willing to make a proportional hazards assumption. There is one added complication: when estimating the first model, one wants to use the data set that would have occured if the subjects being followed for state 1 had not had an artificial censoring, that is, had continued to be followed for event 1 even after event 2 occured. Sometimes this can be filled in directly, e.g., if we knew the enrollment dates for each subject along with the date that follow-up for the study was terminated, and there was no lost to follow-up (only administrative censoring.) An example is the mgus2 data set, where follow-up for death continued after the occurence of plasma cell malignancy. In practice what is done is to estimate the overall censoring distribution and give subjects artificial follow-up.

The function below creates a data set that can then be used with coxph.

⟨*finegray*⟩=

```
finegray <- function(formula, data, weights, subset, na.action= na.pass,
                     etype, prefix="fg", count="", id, timefix=TRUE) {
    Call <- match.call()
    indx <- match(c("formula", "data", "weights", "subset", "id"),
```

```
          names(Call), nomatch=0)
if (indx[1] ==0) stop("A formula argument is required")
temp <- Call[c(1,indx)]  # only keep the arguments we wanted
temp$na.action <- na.action
temp[[1L]] <- quote(stats::model.frame)  # change the function called

special <- c("strata", "cluster")
temp$formula <- if(missing(data)) terms(formula, special)
else              terms(formula, special, data=data)

mf <- eval(temp, parent.frame())
if (nrow(mf) ==0) stop("No (non-missing) observations")
Terms <- terms(mf)

Y <- model.extract(mf, "response")
if (!inherits(Y, "Surv")) stop("Response must be a survival object")
type <- attr(Y, "type")
if (type!='mright' && type!='mcounting')
    stop("Fine-Gray model requires a multi-state survival")
nY <- ncol(Y)
states <- attr(Y, "states")
if (timefix) Y <- aeqSurv(Y)

strats <- attr(Terms, "specials")$strata
if (length(strats)) {
    stemp <- untangle.specials(Terms, 'strata', 1)
    if (length(stemp$vars)==1) strata <- mf[[stemp$vars]]
    else strata <- survival::strata(mf[,stemp$vars], shortlabel=TRUE)
    istrat <- as.numeric(strata)
    mf[stemp$vars] <- NULL
    }
else istrat <- rep(1, nrow(mf))

id <- model.extract(mf, "id")
if (!is.null(id)) mf["(id)"] <- NULL  # don't leave it in result
user.weights <- model.weights(mf)
if (is.null(user.weights)) user.weights <- rep(1.0, nrow(mf))

cluster<- attr(Terms, "specials")$cluster
if (length(cluster)) {
    stop("a cluster() term is not valid")
}

# If there is start-stop data, then there needs to be an id
#  also check that this is indeed a competing risks form of data.
# Mark the first and last obs of each subject, as we need it later.
```

```
    #  Observations may not be in time order within a subject
    delay <- FALSE  # is there delayed entry?
    if (type=="mcounting") {
        if (is.null(id)) stop("(start, stop] data requires a subject id")
        else {
            index <- order(id, Y[,2]) # by time within id
            sorty <- Y[index,]
            first <- which(!duplicated(id[index]))
            last  <- c(first[-1] -1, length(id))
            if (any(sorty[-last, 3] != 0))
                stop("a subject has a transition before their last time point")
            delta <- c(sorty[-1,1], 0) - sorty[,2]
            if (any(delta[-last] !=0))
                stop("a subject has gaps in time")
            if (any(Y[first,1] > min(Y[,2]))) delay <- TRUE
            temp1 <- temp2 <- rep(FALSE, nrow(mf))
            temp1[index[first]] <- TRUE
            temp2[index[last]]  <- TRUE
            first <- temp1  #used later
            last <-  temp2
         }
    } else last <- rep(TRUE, nrow(mf))

    if (missing(etype)) enum <- 1  #generate a data set for which endpoint?
    else {
        index <- match(etype, states)
        if (any(is.na(index)))
            stop ("etype argument has a state that is not in the data")
        enum <- index[1]
        if (length(index) > 1) warning("only the first endpoint was used")
    }

    # make sure count, if present is syntactically valid
    if (!missing(count)) count <- make.names(count) else count <- NULL
    oname <- paste0(prefix, c("start", "stop", "status", "wt"))

    ⟨finegray-censor⟩
    ⟨finegray-build⟩
}
```

The censoring and truncation distributions are

$$G(t) = \prod_{s \le t} \left(1 - \frac{c(s)}{r_c(s)}\right)$$
$$H(t) = \prod_{s > t} \left(1 - \frac{e(s)}{r_e(s)}\right)$$

where $c(t)$ is the number of subjects censored at time t, $e(t)$ is the number who enter at time t, and r is the size of the relevant risk set. These are equations 5 and 6 of Geskus (Biometrics 2011). Note that both G and H are right continuous functions. For tied times the assumption is that event $<$ censor $<$ entry. For G we use a modified Kapan-Meier where any events at censoring time t are removed from the risk set just before time t. To avoid issues with times that are nearly identical (but not quite) we first convert to an integer time scale, and then move events backwards by .2. Since this is a competing risks data set any non-censored observation for a subject is their last, so this time shift does not goof up the alignment of start, stop data. For the truncation distribution it is the subjects with times at or before time t that are in the risk set $r_e(t)$ for truncation at (or before) t. H can be calculated using an ordinary KM on the reverse time scale.

When there is (start,stop) data and hence multiple observations per subject, calculation of G needs use a status that is 1 only for the *last* row row of a censored subject.

⟨*finegray-censor*⟩=
```
 if (ncol(Y) ==2) {
     temp <- min(Y[,1], na.rm=TRUE)
     if (temp >0) zero <- 0
     else zero <- 2*temp -1  # a value less than any observed y
     Y <- cbind(zero, Y)  # add a start column
 }

 utime <- sort(unique(c(Y[,1:2])))  # all the unique times
 newtime <- matrix(findInterval(Y[,1:2], utime), ncol=2)
 status <- Y[,3]

 newtime[status !=0, 2] <- newtime[status !=0,2] - .2
 Gsurv <- survfit(Surv(newtime[,1], newtime[,2], last & status==0) ~ istrat,
                  se.fit=FALSE)
```

The calculation for H is also done on the integer scale. Otherwise we will someday be clobbered by times that differ only in round off error. The only nuisance is the status variable, which is 1 for the first row of each subject, since the data set may not be in sorted order. The offset of .2 used above is not needed, but due to the underlying integer scale it doesn't harm anything either. Reversal of the time scale leads to a left continuous function which we fix up later.

⟨*finegray-censor*⟩=
```
 if (delay)
```

```
Hsurv <- survfit(Surv(-newtime[,2], -newtime[,1], first) ~ istrat,
                 se.fit =FALSE)
```

Consider the following data set:

- Events of type 1 at times 1, 4, 5, 10
- Events of type 2 at times 2, 5, 8
- Censors at times 3, 4, 4, 6, 8, 9, 12

The censoring distribution will have the following shape:

interval	(0,3]	(3,4]	(4,6]	(6,8]	(8,12]	12+
C(t)	1	11/12	(11/12)(8/10)	(11/15)(5/6)	(11/15)(5/6)(3/4)	0
	1.0000	.9167	.7333	.6111	.4583	

Notice that the event at time 4 is not counted in the risk set at time 4, so the jump is 8/10 rather than 8/11. Likewise at time 8 the risk set has 4 instead of 5: censors occur after deaths.

When creating the data set for event type 1, subjects who have an event of type 2 get extended out using this censoring distribution. The event at time 2, for instance, appears as a censored observation with time dependent weights of $G(t)$. The type 2 event at time 5 has weight 1 up through time 5, then weights of $G(t)/C(5)$ for the remainder. This means a weight of 1 over (5,6], 5/6 over (6,8], (5/6)(3/4) over (9,12] and etc.

Though there are 6 unique censoring intervals, in the created data set for event type 1 we only need to know case weights at times 1, 4, 5, and 10; the information from the (4,6] and (6,8] intervals will never be used. To create a minimal sized data set we can leave those intervals out. $G(t)$ only drops to zero if the largest time(s) are censored observations, so by definition no events lie in an interval with $G(t) = 0$.

If there is delayed entry, then the set of intervals is larger due to a merge with the jumps in Hsurv. The truncation distribution Hsurv (H) will become 0 at the first entry time; it is a left continuous function whereas Gsurv (G) is right continuous. We can slide H one point to the left and merge them at the jump points.

⟨*finegray-build*⟩=
```
status <- Y[, 3]

# Do computations separately for each stratum
stratfun <- function(i) {
    keep <- (istrat ==i)
    times <- sort(unique(Y[keep & status == enum, 2])) #unique event times
    if (length(times)==0) return(NULL)  #no events in this stratum
    tdata <- mf[keep, -1, drop=FALSE]
    maxtime <- max(Y[keep, 2])

    Gtemp <- Gsurv[i]
    if (delay) {
        Htemp <- Hsurv[i]
        dtime <- rev(-Htemp$time[Htemp$n.event > 0])
```

```
        dprob <- c(rev(Htemp$surv[Htemp$n.event > 0])[-1], 1)
        ctime <- Gtemp$time[Gtemp$n.event > 0]
        cprob <- c(1, Gtemp$surv[Gtemp$n.event > 0])
        temp <- sort(unique(c(dtime, ctime))) # these will all be integers
        index1 <- findInterval(temp, dtime)
        index2 <- findInterval(temp, ctime)
        ctime <- utime[temp]
        cprob <- dprob[index1] * cprob[index2+1]  # G(t)H(t), eq 11 Geskus
    }
    else {
        ctime <- utime[Gtemp$time[Gtemp$n.event > 0]]
        cprob <- Gtemp$surv[Gtemp$n.event > 0]
    }

    ct2 <- c(ctime, maxtime)
    cp2 <- c(1.0, cprob)
    index <- findInterval(times, ct2, left.open=TRUE)
    index <- sort(unique(index))  # the intervals that were actually seen
    # times before the first ctime get index 0, those between 1 and 2 get 1
    ckeep <- rep(FALSE, length(ct2))
    ckeep[index] <- TRUE
    expand <- (Y[keep, 3] !=0 & Y[keep,3] != enum & last[keep]) #which rows to expand
    split <- .Call(Cfinegray, Y[keep,1], Y[keep,2], ct2, cp2, expand,
                   c(TRUE, ckeep))
    tdata <- tdata[split$row,,drop=FALSE]
    tstat <- ifelse((status[keep])[split$row]== enum, 1, 0)


    tdata[[oname[1]]] <- split$start
    tdata[[oname[2]]] <- split$end
    tdata[[oname[3]]] <- tstat
    tdata[[oname[4]]] <- split$wt * user.weights[split$row]
    if (!is.null(count)) tdata[[count]] <- split$add
    tdata
}

if (max(istrat) ==1) result <- stratfun(1)
else {
    tlist <- lapply(1:max(istrat), stratfun)
    result <- do.call("rbind", tlist)
}

rownames(result) <- NULL   #remove all the odd labels that R adds
attr(result, "event") <- states[enum]
result
```

3.1 The predict method

The `predict.coxph` function produces various types of predicted values from a Cox model. The arguments are

object The result of a call to `coxph`.

newdata Optionally, a new data set for which prediction is desired. If this is absent predictions are for the observations used fit the model.

type The type of prediction

- lp = the linear predictor for each observation
- risk = the risk score $exp(lp)$ for each observation
- expected = the expected number of events
- survival = predicted survival = exp(-expected)
- terms = a matrix with one row per subject and one column for each term in the model.

se.fit Whether or not to return standard errors of the predictions.

na.action What to do with missing values *if* there is new data.

terms The terms that are desired. This option is almost never used, so rarely in fact that it's hard to justify keeping it.

collapse An optional vector of subject identifiers, over which to sum or 'collapse' the results

reference the reference context for centering the results

... All predict methods need to have a ... argument; we make no use of it however.

The first task of the routine is to reconsruct necessary data elements that were not saved as a part of the `coxph` fit. We will need the following components:

- for type='expected' residuals we need the orignal survival y. This is saved in coxph objects by default so will only need to be fetched in the highly unusual case that a user specfied `y=FALSE` in the orignal call.
- for any call with either newdata, standard errors, or type='terms' the original X matrix, weights, strata, and offset. When checking for the existence of a saved X matrix we can't use `object$x` since that will also match the `xlevels` component.
- the new data matrix, if any

⟨*predict.coxph*⟩=

```
 predict.coxph <- function(object, newdata,
                           type=c("lp", "risk", "expected", "terms", "survival"),
                           se.fit=FALSE, na.action=na.pass,
                           terms=names(object$assign), collapse,
```

```
                    reference=c("strata", "sample", "zero"), ...) {
    ⟨pcoxph-init⟩
    ⟨pcoxph-getdata⟩
    if (type=="expected") {
        ⟨pcoxph-expected⟩
        }
    else {
        ⟨pcoxph-simple⟩
        ⟨pcoxph-terms⟩
        }
    ⟨pcoxph-finish⟩
    }
```

We start of course with basic argument checking. Then retrieve the model parameters: does it have a strata statement, offset, etc. The `Terms2` object is a model statement without the strata or cluster terms, appropriate for recreating the matrix of covariates X. For type=expected the response variable needs to be kept, if not we remove it as well since the user's newdata might not contain one. The type= survival is treated the same as type expected.

⟨*pcoxph-init*⟩=

```
 if (!inherits(object, 'coxph'))
     stop("Primary argument much be a coxph object")

 Call <- match.call()
 type <-match.arg(type)
 if (type=="survival") {
     survival <- TRUE
     type <- "expected"  #this is to stop lots of "or" statements
 }
 else survival <- FALSE

 n <- object$n
 Terms <-  object$terms

 if (!missing(terms)) {
     if (is.numeric(terms)) {
         if (any(terms != floor(terms) |
                 terms > length(object$assign) |
                 terms <1)) stop("Invalid terms argument")
         }
     else if (any(is.na(match(terms, names(object$assign)))))
        stop("a name given in the terms argument not found in the model")
     }

 # I will never need the cluster argument, if present delete it.
 #  Terms2 are terms I need for the newdata (if present), y is only
 #  needed there if type == 'expected'
```

```
if (length(attr(Terms, 'specials')$cluster)) {
    temp <- untangle.specials(Terms, 'cluster', 1)
    Terms  <- object$terms[-temp$terms]
    }
else Terms <- object$terms

if (type != 'expected') Terms2 <- delete.response(Terms)
else Terms2 <- Terms

has.strata <- !is.null(attr(Terms, 'specials')$strata)
has.offset <- !is.null(attr(Terms, 'offset'))
has.weights <- any(names(object$call) == 'weights')
na.action.used <- object$na.action
n <- length(object$residuals)

if (missing(reference) && type=="terms") reference <- "sample"
else reference <- match.arg(reference)
```

The next task of the routine is to reconsruct necessary data elements that were not saved as a part of the `coxph` fit. We will need the following components:

- for type='expected' residuals we need the orignal survival y. This is saved in coxph objects by default so will only need to be fetched in the highly unusual case that a user specfied `y=FALSE` in the orignal call. We also need the strata in this case. Grabbing it is the same amount of work as grabbing X, so gets lumped with that case in the code.
- for any call with either standard errors, reference strata, or type='terms' the original X matrix, weights, strata, and offset. When checking for the existence of a saved X matrix we can't use `object$x` since that will also match the `xlevels` component.
- the new data matrix, if present, along with offset and strata.

For the case that none of the above are needed, we can use the `linear.predictors` component of the fit. The variable `use.x` signals this case, which takes up almost none of the code but is common in usage.

The check below that nrow(mf)==n is to avoid data sets that change under our feet. A fit was based on data set "x", and when we reconstruct the data frame it is a different size! This means someone changed the data between the model fit and the extraction of residuals. One other non-obvious case is that coxph treats the model `age:strata(grp)` as though it were `age:strata(grp) + strata(grp)`. The untangle.specials function will return `vars= strata(grp), terms=integer(0)`; the first shows a strata to extract and the second that there is nothing to remove from the terms structure.

⟨*pcoxph-getdata*⟩=
```
have.mf <- FALSE
if (type == "expected") {
    y <- object[['y']]
    if (is.null(y)) {  # very rare case
```

```
        mf <- stats::model.frame(object)
        y <-  model.extract(mf, 'response')
        have.mf <- TRUE  #for the logic a few lines below, avoid double work
        }
    }

# This will be needed if there are strata, and is cheap to compute
strat.term <- untangle.specials(Terms, "strata")
if (se.fit || type=='terms' || (!missing(newdata) && type=="expected") ||
    (has.strata && (reference=="strata") || type=="expected")) {
    use.x <- TRUE
    if (is.null(object[['x']]) || has.weights || has.offset ||
         (has.strata && is.null(object$strata))) {
        # I need the original model frame
        if (!have.mf) mf <- stats::model.frame(object)
        if (nrow(mf) != n)
            stop("Data is not the same size as it was in the original fit")
        x <- model.matrix(object, data=mf)
        if (has.strata) {
            if (!is.null(object$strata)) oldstrat <- object$strata
            else {
                if (length(strat.term$vars)==1) oldstrat <- mf[[strat.term$vars]]
                else oldstrat <- strata(mf[,strat.term$vars], shortlabel=TRUE)
              }
        }
        else oldstrat <- rep(0L, n)

        weights <- model.weights(mf)
        if (is.null(weights)) weights <- rep(1.0, n)
        offset <- model.offset(mf)
        if (is.null(offset))  offset  <- 0
    }
    else {
        x <- object[['x']]
        if (has.strata) oldstrat <- object$strata
        else oldstrat <- rep(0L, n)
        weights <-  rep(1.,n)
        offset <-   0
    }
}
else {
    # I won't need strata in this case either
    if (has.strata) {
        stemp <- untangle.specials(Terms, 'strata', 1)
        Terms2  <- Terms2[-stemp$terms]
        has.strata <- FALSE  #remaining routine never needs to look
```

```
    }
    oldstrat <- rep(0L, n)
    offset <- 0
    use.x <- FALSE
}
```

Now grab data from the new data set. We want to use model.frame processing, in order to correctly expand factors and such. We don't need weights, however, and don't want to make the user include them in their new dataset. Thus we build the call up the way it is done in coxph itself, but only keeping the newdata argument. Note that terms2 may have fewer variables than the original model: no cluster and if type!= expected no response. If the original model had a strata, but newdata does not, we need to remove the strata from xlev to stop a spurious warning message.

⟨*pcoxph-getdata*⟩=

```
if (!missing(newdata)) {
    use.x <- TRUE  #we do use an X matrix later
    tcall <- Call[c(1, match(c("newdata", "collapse"), names(Call), nomatch=0))]
    names(tcall)[2] <- 'data'  #rename newdata to data
    tcall$formula <- Terms2  #version with no response
    tcall$na.action <- na.action #always present, since there is a default
    tcall[[1L]] <- quote(stats::model.frame)  # change the function called

    if (!is.null(attr(Terms, "specials")$strata) && !has.strata) {
       temp.lev <- object$xlevels
       temp.lev[[strat.term$vars]] <- NULL
       tcall$xlev <- temp.lev
    }
    else tcall$xlev <- object$xlevels
    mf2 <- eval(tcall, parent.frame())

    collapse <- model.extract(mf2, "collapse")
    n2 <- nrow(mf2)

    if (has.strata) {
        if (length(strat.term$vars)==1) newstrat <- mf2[[strat.term$vars]]
        else newstrat <- strata(mf2[,strat.term$vars], shortlabel=TRUE)
        if (any(is.na(match(newstrat, oldstrat))))
            stop("New data has a strata not found in the original model")
        else newstrat <- factor(newstrat, levels=levels(oldstrat)) #give it all
        if (length(strat.term$terms))
            newx <- model.matrix(Terms2[-strat.term$terms], mf2,
                        contr=object$contrasts)[,-1,drop=FALSE]
        else newx <- model.matrix(Terms2, mf2,
                        contr=object$contrasts)[,-1,drop=FALSE]
         }
```

```
    else {
        newx <- model.matrix(Terms2, mf2,
                        contr=object$contrasts)[,-1,drop=FALSE]
        newstrat <- rep(0L, nrow(mf2))
        }

    newoffset <- model.offset(mf2)
    if (is.null(newoffset)) newoffset <- 0
    if (type== 'expected') {
        newy <- model.response(mf2)
        if (attr(newy, 'type') != attr(y, 'type'))
            stop("New data has a different survival type than the model")
        }
    na.action.used <- attr(mf2, 'na.action')
    }
else n2 <- n
```

When we do not need standard errors the computation of expected hazard is very simple since the martingale residual is defined as status - expected. The 0/1 status is saved as the last column of y.

⟨*pcoxph-expected*⟩=
```
if (missing(newdata))
    pred <- y[,ncol(y)] - object$residuals
if (!missing(newdata) || se.fit) {
    ⟨pcoxph-expected2⟩
    }
if (survival) { #it actually was type= survival, do one more step
    if (se.fit) se <- se * exp(-pred)
    pred <- exp(-pred)  # probablility of being in state 0
}
```

The more general case makes use of the [agsurv] routine to calculate a survival curve for each strata. The routine is defined in the section on individual Cox survival curves. The code here closely matches that. The routine only returns values at the death times, so we need approx to get a complete index.

One non-obvious, but careful choice is to use the residuals for the predicted value instead of the compuation below, whenever operating on the original data set. This is a consequence of the Efron approx. When someone in a new data set has exactly the same time as one of the death times in the original data set, the code below implicitly makes them the "last" death in the set of tied times. The Efron approx puts a tie somewhere in the middle of the pack. This is way too hard to work out in the code below, but thankfully the original Cox model already did it. However, it does mean that a different answer will arise if you set newdata = the original coxph data set. Standard errors have the same issue, but 1. they are hardly used and 2. the original coxph doesn't do that calculation. So we do what's easiest.

⟨*pcoxph-expected2*⟩=
```
ustrata <- unique(oldstrat)
```

```
risk <- exp(object$linear.predictors)
x <- x - rep(object$means, each=nrow(x))  #subtract from each column
if (missing(newdata)) #se.fit must be true
    se <- double(n)
else {
    pred <- se <- double(nrow(mf2))
    newx <- newx - rep(object$means, each=nrow(newx))
    newrisk <- c(exp(newx %*% object$coef) + newoffset)
    }

survtype<- ifelse(object$method=='efron', 3,2)
for (i in ustrata) {
    indx <- which(oldstrat == i)
    afit <- agsurv(y[indx,,drop=F], x[indx,,drop=F],
                                  weights[indx], risk[indx],
                                  survtype, survtype)
    afit.n <- length(afit$time)
    if (missing(newdata)) {
        # In this case we need se.fit, nothing else
        j1 <- approx(afit$time, 1:afit.n, y[indx,1], method='constant',
                     f=0, yleft=0, yright=afit.n)$y
        chaz <- c(0, afit$cumhaz)[j1 +1]
        varh <- c(0, cumsum(afit$varhaz))[j1 +1]
        xbar <- rbind(0, afit$xbar)[j1+1,,drop=F]
        if (ncol(y)==2) {
            dt <- (chaz * x[indx,]) - xbar
            se[indx] <- sqrt(varh + rowSums((dt %*% object$var) *dt)) *
                risk[indx]
            }
        else {
            j2 <- approx(afit$time, 1:afit.n, y[indx,2], method='constant',
                     f=0, yleft=0, yright=afit.n)$y
            chaz2 <- c(0, afit$cumhaz)[j2 +1]
            varh2 <- c(0, cumsum(afit$varhaz))[j2 +1]
            xbar2 <- rbind(0, afit$xbar)[j2+1,,drop=F]
            dt <- (chaz * x[indx,]) - xbar
            v1 <- varh +  rowSums((dt %*% object$var) *dt)
            dt2 <- (chaz2 * x[indx,]) - xbar2
            v2 <- varh2 + rowSums((dt2 %*% object$var) *dt2)
            se[indx] <- sqrt(v2-v1)* risk[indx]
            }
        }

    else {
        #there is new data
        use.x <- TRUE
```

```
        indx2 <- which(newstrat == i)
        j1 <- approx(afit$time, 1:afit.n, newy[indx2,1],
                     method='constant', f=0, yleft=0, yright=afit.n)$y
        chaz <-c(0, afit$cumhaz)[j1+1]
        pred[indx2] <- chaz * newrisk[indx2]
        if (se.fit) {
            varh <- c(0, cumsum(afit$varhaz))[j1+1]
            xbar <- rbind(0, afit$xbar)[j1+1,,drop=F]
            }
        if (ncol(y)==2) {
            if (se.fit) {
                dt <- (chaz * newx[indx2,]) - xbar
                se[indx2] <- sqrt(varh + rowSums((dt %*% object$var) *dt)) *
                    newrisk[indx2]
                }
            }
        else {
            j2 <- approx(afit$time, 1:afit.n, newy[indx2,2],
                     method='constant', f=0, yleft=0, yright=afit.n)$y
                        chaz2 <- approx(-afit$time, afit$cumhaz, -newy[indx2,2],
                      method="constant", rule=2, f=0)$y
            chaz2 <-c(0, afit$cumhaz)[j2+1]
            pred[indx2] <- (chaz2 - chaz) * newrisk[indx2]

            if (se.fit) {
                varh2 <- c(0, cumsum(afit$varhaz))[j1+1]
                xbar2 <- rbind(0, afit$xbar)[j1+1,,drop=F]
                dt <- (chaz * newx[indx2,]) - xbar
                dt2 <- (chaz2 * newx[indx2,]) - xbar2

                v2 <- varh2 + rowSums((dt2 %*% object$var) *dt2)
                v1 <- varh +  rowSums((dt %*% object$var) *dt)
                se[indx2] <- sqrt(v2-v1)* risk[indx2]
                }
            }
        }
    }
```

For these three options what is returned is a *relative* prediction which compares each observation to the average for the data set. Partly this is practical. Say for instance that a treatment covariate was coded as 0=control and 1=treatment. If the model were refit using a new coding of 3=control 4=treatment, the results of the Cox model would be exactly the same with respect to coefficients, variance, tests, etc. The raw linear predictor $X\beta$ however would change, increasing by a value of 3β. The relative predictor

$$\eta_i = X_i\beta - (1/n)\sum_j X_j\beta \tag{7}$$

will stay the same. The second reason for doing this is that the Cox model is a relative risks model rather than an absolute risks model, and thus relative predictions are almost certainly what the user was thinking of.

When the fit was for a stratified Cox model more care is needed. For instance assume that we had a fit that was stratified by sex with covaritate x, and a second data set were created where for the females x is replaced by $x + 3$. The Cox model results will be unchanged for the two models, but the 'normalized' linear predictors $(x - \overline{x})'\beta$ will not be the same. This reflects a more fundamental issue that the for a stratified Cox model relative risks are well defined only *within* a stratum, i.e. for subject pairs that share a common baseline hazard. The example above is artificial, but the problem arises naturally whenever the model includes a strata by covariate interaction. So for a stratified Cox model the predictions should be forced to sum to zero within each stratum, or equivalently be made relative to the weighted mean of the stratum. Unfortunately, this important issue was not realized until late in 2009 when a puzzling query was sent to the author involving the results from such an interaction. Note that this issue did not arise with type='expected', which has a natural scaling.

An offset variable, if specified, is treated like any other covariate with respect to centering. The logic for this choice is not as compelling, but it seemed the best that I could do. Note that offsets play no role whatever in predicted terms, only in the lp and risk.

Start with the simple ones

⟨*pcoxph-simple*⟩=
```
 if (is.null(object$coefficients))
     coef<-numeric(0)
 else {
     # Replace any NA coefs with 0, to stop NA in the linear predictor
     coef <- ifelse(is.na(object$coefficients), 0, object$coefficients)
     }

 if (missing(newdata)) {
     offset <- offset - mean(offset)
     if (has.strata && reference=="strata") {
         # We can't use as.integer(oldstrat) as an index, if oldstrat is
         #   a factor variable with unrepresented levels as.integer could
         #   give 1,2,5 for instance.
         xmeans <- rowsum(x*weights, oldstrat)/c(rowsum(weights, oldstrat))
         newx <- x - xmeans[match(oldstrat,row.names(xmeans)),]
     }
     else if (use.x) {
         if (reference == "zero") newx <- x
         else newx <- x - rep(object$means, each=nrow(x))
     }
 }
 else {
     offset <- newoffset - mean(offset)
     if (has.strata && reference=="strata") {
         xmeans <- rowsum(x*weights, oldstrat)/c(rowsum(weights, oldstrat))
```

```
        newx <- newx - xmeans[match(newstrat, row.names(xmeans)),]
        }
    else if (reference!= "zero")
        newx <- newx - rep(object$means, each=nrow(newx))
    }

if (type=='lp' || type=='risk') {
    if (use.x) pred <- drop(newx %*% coef) + offset
    else pred <- object$linear.predictors
    if (se.fit) se <- sqrt(rowSums((newx %*% object$var) *newx))

    if (type=='risk') {
        pred <- exp(pred)
        if (se.fit) se <- se * sqrt(pred)  # standard Taylor series approx
        }
    }
```

The type=terms residuals are a bit more work. In Splus this code used the Build.terms function, which was essentially the code from predict.lm extracted out as a separate function. As of March 2010 (today) a check of the Splus function and the R code for predict.lm revealed no important differences. A lot of the bookkeeping in both is to work around any possible NA coefficients resulting from a singularity. The basic formula is to

1. If the model has an intercept, then sweep the column means out of the X matrix. We've already done this.
2. For each term separately, get the list of coefficients that belong to that term; call this list `tt`.
3. Restrict X, β and V (the variance matrix) to that subset, then the linear predictor is $X\beta$ with variance matrix XVX'. The standard errors are the square root of the diagonal of this latter matrix. This can be computed, as colSums((X

Note that the `assign` component of a coxph object is the same as that found in Splus models (a list), most R models retain a numeric vector which contains the same information but it is not as easily used. The first first part of predict.lm in R rebuilds the list form as its `asgn` variable. I can skip this part since it is already done.

⟨*pcoxph-terms*⟩=
```
 else if (type=='terms') {
    asgn <- object$assign
    nterms<-length(asgn)
    pred<-matrix(ncol=nterms,nrow=NROW(newx))
    dimnames(pred) <- list(rownames(newx), names(asgn))
    if (se.fit) se <- pred

    for (i in 1:nterms) {
        tt <- asgn[[i]]
```

```
        tt <- tt[!is.na(object$coefficients[tt])]
        xtt <- newx[,tt, drop=F]
        pred[,i] <- xtt %*% object$coefficient[tt]
        if (se.fit)
            se[,i] <- sqrt(rowSums((xtt %*% object$var[tt,tt]) *xtt))
        }
    pred <- pred[,terms, drop=F]
    if (se.fit) se <- se[,terms, drop=F]

    attr(pred, 'constant') <- sum(object$coefficients*object$means, na.rm=T)
    }
```

To finish up we need to first expand out any missings in the result based on the na.action, and optionally collapse the results within a subject. What should we do about the standard errors when collapse is specified? We assume that the individual pieces are independent and thus var(sum) = sum(variances). The statistical justification of this is quite solid for the linear predictor, risk and terms type of prediction due to independent increments in a martingale. For expecteds the individual terms are positively correlated so the se will be too small. One solution would be to refuse to return an se in this case, but the the bias should usually be small, and besides it would be unkind to the user.

Prediction of type='terms' is expected to always return a matrix, or the R termplot() function gets unhappy.

⟨*pcoxph-finish*⟩=
```
 if (type != 'terms') {
     pred <- drop(pred)
     if (se.fit) se <- drop(se)
     }

 if (!is.null(na.action.used)) {
     pred <- napredict(na.action.used, pred)
     if (is.matrix(pred)) n <- nrow(pred)
     else                 n <- length(pred)
     if(se.fit) se <- napredict(na.action.used, se)
     }

 if (!missing(collapse) && !is.null(collapse)) {
     if (length(collapse) != n2) stop("Collapse vector is the wrong length")
     pred <- rowsum(pred, collapse)  # in R, rowsum is a matrix, always
     if (se.fit) se <- sqrt(rowsum(se^2, collapse))
     if (type != 'terms') {
         pred <- drop(pred)
         if (se.fit) se <- drop(se)
         }
     }
```

```
if (se.fit) list(fit=pred, se.fit=se)
else pred
```

4 Concordance

4.1 Main routine

The concordance statistic is the most used measure of goodness-of-fit in survival models. In general let y_i and x_i be observed and predicted data values. A pair of obervations i, j is considered condordant if either $y_i > y_j, x_i > x_j$ or $y_i < y_j, x_i < x_j$. The concordance is the fraction of concordant pairs. For a Cox model remember that the predicted survival $\hat{y}$ is longer if the risk score $X\beta$ is lower, so we have to flip the definition and count "discordant" pairs, this is done at the end of the routine.

One wrinkle is what to do with ties in either y or x. Such pairs can be ignored in the count (treated as incomparable), treated as discordant, or given a score of 1/2.

- Kendall's τ-a scores ties as 0.
- Kendall's τ-b and the Goodman-Kruskal γ ignore ties in either y or x.
- Somers' d treats ties in y as incomparable, pairs that are tied in x (but not y) score as 1/2. The AUC from logistic regression is equal to Somers' d.

All three of the above range from -1 to 1, the concordance is $(d + 1)/2$. For survival data any pairs which cannot be ranked with certainty are considered incomparable. For instance y_i is censored at time 10 and y_j is an event (or censor) at time 20. Subject i may or may not survive longer than subject j. Note that if y_i is censored at time 10 and y_j is an event at time 10 then $y_i > y_j$. Observations that are in different strata are also incomparable, since the Cox model only compares within strata.

The program creates 4 variables, which are the number of concordant pairs, discordant, tied on time, and tied on x but not on time. The default concordance is based on the Somers'/AUC definition, but all 4 values are reported back so that a user can recreate Kendall's or Goodmans values if desired.

Here is the main routine.

⟨*concordance*⟩=
```
concordance <- function(object, ...)
    UseMethod("concordance")

concordance.formula <- function(object, data,
                                weights, subset, na.action, cluster,
                                ymin, ymax,
                                timewt=c("n", "S", "S/G", "n/G", "n/G2", "I"),
                                influence=0, ranks=FALSE, reverse=FALSE,
                                timefix=TRUE, keepstrata=10, ...) {
    Call <- match.call()  # save a copy of of the call, as documentation
    timewt <- match.arg(timewt)
```

```
    if (missing(ymin)) ymin <- NULL
    if (missing(ymax)) ymax <- NULL

    index <- match(c("data", "weights", "subset", "na.action",
                     "cluster"),
                   names(Call), nomatch=0)
    temp <- Call[c(1, index)]
    temp[[1L]] <-  quote(stats::model.frame)
    special <- c("strata", "cluster")
    temp$formula <- if(missing(data)) terms(object, special)
                    else              terms(object, special, data=data)
    mf <- eval(temp, parent.frame())  # model frame
    if (nrow(mf) ==0) stop("No (non-missing) observations")
    Terms <- terms(mf)

    Y <- model.response(mf)
    if (inherits(Y, "Surv")) {
        if (timefix) Y <- aeqSurv(Y)
    } else {
        if (is.factor(Y) && (is.ordered(Y) || length(levels(Y))==2))
            Y <- Surv(as.numeric(Y))
        else if (is.numeric(Y) && is.vector(Y))  Y <- Surv(Y)
        else stop("left hand side of the formula must be a numeric vector,
survival object, or an orderable factor")
        if (timefix) Y <- aeqSurv(Y)
    }
    n <- nrow(Y)

    wt <- model.weights(mf)
    offset<- attr(Terms, "offset")
    if (length(offset)>0) stop("Offset terms not allowed")

    stemp <- untangle.specials(Terms, "strata")
    if (length(stemp$vars)) {
        if (length(stemp$vars)==1) strat <- mf[[stemp$vars]]
        else strat <- strata(mf[,stemp$vars], shortlabel=TRUE)
        Terms <- Terms[-stemp$terms]
    }
    else strat <- NULL

    # if "cluster" was an argument, use it, otherwise grab it from the model
    group <- model.extract(mf, "cluster")
    cluster<- attr(Terms, "specials")$cluster
    if (length(cluster)) {
        tempc <- untangle.specials(Terms, 'cluster', 1:10)
        ord <- attr(Terms, 'order')[tempc$terms]
```

```
        if (any(ord>1)) stop ("Cluster can not be used in an interaction")
        cluster <- strata(mf[,tempc$vars], shortlabel=TRUE)  #allow multiples
        Terms <- Terms[-tempc$terms]  # toss it away
    }
    if (length(group)) cluster <- group

    x <- model.matrix(Terms, mf)[,-1, drop=FALSE]  #remove the intercept
    if (ncol(x) > 1) stop("Only one predictor variable allowed")

    if (!is.null(ymin) & (length(ymin)> 1 || !is.numeric(ymin)))
        stop("ymin must be a single number")
    if (!is.null(ymax) & (length(ymax)> 1 || !is.numeric(ymax)))
        stop("ymax must be a single number")
    if (!is.logical(reverse))
        stop ("the reverse argument must be TRUE/FALSE")

    fit <- concordancefit(Y, x, strat, wt, ymin, ymax, timewt, cluster,
                          influence, ranks, reverse, keepstrata=keepstrata)
    na.action <- attr(mf, "na.action")
    if (length(na.action)) fit$na.action <- na.action
    fit$call <- Call

    class(fit) <- 'concordance'
    fit
}

print.concordance <- function(x, digits= max(1L, getOption("digits") - 3L),
                              ...) {
    if(!is.null(cl <- x$call)) {
        cat("Call:\n")
        dput(cl)
        cat("\n")
        }
    omit <- x$na.action
    if(length(omit))
        cat("n=", x$n, " (", naprint(omit), ")\n", sep = "")
    else cat("n=", x$n, "\n")

    if (length(x$concordance) > 1) {
        # result of a call with multiple fits
        tmat <- cbind(concordance= x$concordance, se=sqrt(diag(x$var)))
        print(round(tmat, digits=digits), ...)
        cat("\n")
    }
    else cat("Concordance= ", format(x$concordance, digits=digits), " se= ",
             format(sqrt(x$var), digits=digits), '\n', sep='')
```

```
    if (!is.matrix(x$count) || nrow(x$count < 11))
        print(round(x$count,2))
    invisible(x)
    }
```

⟨*concordancefit*⟩

⟨*btree*⟩

The concordancefit function is broken out separately, since it is called by all of the methods. It is also called directly by the the `coxph` routine. If y is not a survival quantity, then all of the options for the `timewt` parameter lead to the same result.

⟨*concordancefit*⟩=

```
concordancefit <- function(y, x, strata, weights, ymin=NULL, ymax=NULL,
                           timewt=c("n", "S", "S/G", "n/G", "n/G2", "I"),
                           cluster, influence=0, ranks=FALSE, reverse=FALSE,
                           timefix=TRUE, keepstrata=10, robustse =TRUE) {
    # The coxph program may occassionally fail, and this will kill the C
    #  routine further below.  So check for it.
    if (any(is.na(x)) || any(is.na(y))) return(NULL)
    timewt <- match.arg(timewt)

    if (!robustse) {ranks <- FALSE; influence =0;}

    # these should only occur if something other package calls this routine
    if (!is.Surv(y)) {
        if (is.factor(y) && (is.ordered(y) || length(levels(y))==2))
            y <- Surv(as.numeric(y))
        else if (is.numeric(y) && is.vector(y))  y <- Surv(y)
        else stop("left hand side of the formula must be a numeric vector,
 survival object, or an orderable factor")
        if (timefix) y <- aeqSurv(y)
    }
    n <- length(y)
    if (length(x) != n) stop("x and y are not the same length")
    if (missing(strata) || length(strata)==0) strata <- rep(1L, n)
    if (length(strata) != n)
        stop("y and strata are not the same length")
    if (missing(weights) || length(weights)==0) weights <- rep(1.0, n)
    else if (length(weights) != n) stop("y and weights are not the same length")

    type <- attr(y, "type")
    if (type %in% c("left", "interval"))
        stop("left or interval censored data is not supported")
    if (type %in% c("mright", "mcounting"))
```

```
    stop("multiple state survival is not supported")

nstrat <- length(unique(strata))
if (!is.logical(keepstrata)) {
    if (!is.numeric(keepstrata))
        stop("keepstrat argument must be logical or numeric")
    else keepstrata <- (nstrat <= keepstrata)
}

if (timewt %in% c("n", "I") && nstrat > 10 && !keepstrata) {
    # Special trickery for matched case-control data, where the
    #  number of strata is huge, n per strata is small, and compute
    #  time becomes excessive.  Make the data all one strata, but over
    #  disjoint time intervals
    stemp <- as.numeric(as.factor(strata)) -1
    if (ncol(y) ==3) {
        delta <- 2+ max(y[,2]) - min(y[,1])
        y[,1] <- y[,1] + stemp*delta
        y[,2] <- y[,2] + stemp*delta
    }
    else {
        delta <- max(y[,1]) +2
        m1 <- rep(-1L, nrow(y))
        y <- Surv(m1 + stemp*delta, y[,1] + stemp*delta, y[,2])
    }
    strata <- rep(1L, n)
    nstrat <- 1
}

# This routine is called once per stratum
docount <- function(y, risk, wts, timeopt= 'n', timefix) {
    n <- length(risk)
    # this next line is mostly invoked in stratified logistic, where
    #  only 1 event per stratum occurs.  All time weightings are the same
    # don't waste time even if the user asked for something different
    if (sum(y[,ncol(y)]) <2) timeopt <- 'n'

    sfit <- survfit(y~1, weights=wts, se.fit=FALSE, timefix=timefix)
    etime <- sfit$time[sfit$n.event > 0]
    esurv <- sfit$surv[sfit$n.event > 0]

    if (length(etime)==0) {
        # the special case of a stratum with no events (it happens)
        # No need to do any more work
        return(list(count= rep(0.0, 6), influence=matrix(0.0, n, 5),
                    resid=NULL))
```

```
 }

if (timeopt %in% c("S/G", "n/G", "n/G2")) {
     temp <- y
     temp[,ncol(temp)] <- 1- temp[,ncol(temp)] # switch event/censor
     gfit <- survfit(temp~1, weights=wts, se.fit=FALSE, timefix=timefix)
     # G has the exact same time values as S
     gsurv <- c(1, gfit$surv)  # We want G(t-)
     gsurv <- gsurv[which(sfit$n.event > 0)]
 }

 npair <- (sfit$n.risk- sfit$n.event)[sfit$n.event>0]
 temp  <- ifelse(esurv==0, 0, esurv/npair)  # avoid 0/0
 timewt <- switch(timeopt,
                  "S" =  sum(wts)*temp,
                  "S/G" = sum(wts)* temp/ gsurv,
                  "n" =   rep(1.0, length(npair)),
                  "n/G" = 1/gsurv,
                  "n/G2"= 1/gsurv^2,
                  "I"  =  rep(1.0, length(esurv))
              )
 if (!is.null(ymin)) timewt[etime < ymin] <- 0
 if (!is.null(ymax)) timewt[etime > ymax] <- 0
 timewt <- ifelse(is.finite(timewt), timewt, 0)  # 0 at risk case

 # order the data: reverse time, censors before deaths
 if (ncol(y)==2) {
     sort.stop <- order(-y[,1], y[,2], risk) -1L
 } else {
     sort.stop  <- order(-y[,2], y[,3], risk) -1L   #order by endpoint
     sort.start <- order(-y[,1]) -1L
 }

 # match each prediction score to the unique set of scores
 # (to deal with ties)
 utemp <- match(risk, sort(unique(risk)))
 bindex <- btree(max(utemp))[utemp]

 storage.mode(y) <- "double"  # just in case y is integer
 storage.mode(wts) <- "double"
 if (robustse) {
     if (ncol(y) ==2)
         fit <- .Call(Cconcordance3, y, bindex, wts, rev(timewt),
                      sort.stop, ranks)
     else fit <- .Call(Cconcordance4, y, bindex, wts, rev(timewt),
                       sort.start, sort.stop, ranks)
```

```
        # The C routine gives back an influence matrix which has columns for
        #  concordant, discordant, tied on x but not y, tied on y, and tied
        #  on both x and y.
        dimnames(fit$influence) <- list(NULL,
               c("concordant", "discordant", "tied.x", "tied.y", "tied.xy"))
        if (ranks) {
            if (ncol(y)==2) dtime <- y[y[,2]==1, 1]
            else dtime <- y[y[,3]==1, 2]
            temp <- data.frame(time= sort(dtime), fit$resid)
            names(temp) <- c("time", "rank", "timewt", "casewt", "variance")
            fit$resid <- temp[temp[,3] > 0,]  # don't return zeros
        }
    }
    else {
        if (ncol(y) ==2)
            fit <- .Call(Cconcordance5, y, bindex, wts, rev(timewt),
                         sort.stop)
        else fit <- .Call(Cconcordance6, y, bindex, wts, rev(timewt),
                          sort.start, sort.stop)
    }
    fit
}

if (nstrat < 2) {
    fit <- docount(y, x, weights, timewt, timefix=timefix)
    count2 <- fit$count[1:5]
    vcox <- fit$count[6]
    fit$count <- fit$count[1:5]
    if (robustse) imat <- fit$influence
    if (ranks) resid <- fit$resid
} else {
    strata <- as.factor(strata)
    ustrat <- levels(strata)[table(strata) >0]  #some strata may have 0 obs
    tfit <- lapply(ustrat, function(i) {
        keep <- which(strata== i)
        docount(y[keep,,drop=F], x[keep], weights[keep], timewt,
                timefix=timefix)
    })
    temp <-  t(sapply(tfit, function(x) x$count))
    fit <- list(count = temp[,1:5])
    count2 <- colSums(fit$count)
    if (!keepstrata) fit$count <- count2
    vcox <- sum(temp[,6])
    if (robustse) {
        imat <- do.call("rbind", lapply(tfit, function(x) x$influence))
```

```
        # put it back into data order
        index <- match(1:n, (1:n)[order(strata)])
        imat <- imat[index,]
        if (ranks) {
            nr <- lapply(tfit, function(x) nrow(x$resid))
            resid <- do.call("rbind", lapply(tfit, function(x) x$resid))
            resid$strata <- rep(ustrat, nr)
        }
    }
}

npair <- sum(count2[1:3])
if (!keepstrata && is.matrix(fit$count)) fit$count <- colSums(fit$count)
somer <- (count2[1] - count2[2])/npair
if (robustse) {
    dfbeta <- weights*((imat[,1]- imat[,2])/npair -
                       (somer/npair)* rowSums(imat[,1:3]))
    if (!missing(cluster) && length(cluster)>0) {
        dfbeta <- tapply(dfbeta, cluster, sum)
        dfbeta <- ifelse(is.na(dfbeta),0, dfbeta)  # if cluster is a factor
    }
    var.somer <- sum(dfbeta^2)
    rval <- list(concordance = (somer+1)/2, count=fit$count, n=n,
                 var = var.somer/4, cvar=vcox/(4*npair^2))
    }
else  rval <- list(concordance = (somer+1)/2, count=fit$count, n=n,
                 cvar=vcox/(4*npair^2))
if (is.matrix(rval$count))
    colnames(rval$count) <- c("concordant", "discordant", "tied.x",
                              "tied.y", "tied.xy")
else names(rval$count) <- c("concordant", "discordant", "tied.x", "tied.y",
                       "tied.xy")

if (influence == 1 || influence==3) rval$dfbeta <- dfbeta/2
if (influence >=2) rval$influence <- imat

if (ranks) rval$ranks <- resid
if (reverse) {
    # flip concordant/discordant values but not the labels
    rval$concordance <- 1- rval$concordance
    if (!is.null(rval$dfbeta)) rval$dfbeta <- -rval$dfbeta
    if (!is.null(rval$influence)) {
        rval$influence <- rval$influence[,c(2,1,3,4,5)]
        colnames(rval$influence) <- colnames(rval$influence)[c(2,1,3,4,5)]
    }
    if (is.matrix(rval$count)) {
```

```
            rval$count <- rval$count[, c(2,1,3,4,5)]
            colnames(rval$count) <- colnames(rval$count)[c(2,1,3,4,5)]
        }
        else {
            rval$count <- rval$count[c(2,1,3,4,5)]
            names(rval$count) <- names(rval$count)[c(2,1,3,4,5)]
        }
        if (ranks) rval$ranks$rank <- -rval$ranks$rank
    }

    rval
}
```

4.2 Methods

Methods are defined for lm, survfit, and coxph objects. Detection of strata, weights, or clustering is the main nuisance, since those are not passed back as part of coxph or survreg objects. Glm and lm objects have the model frame by default, but that can be turned off by a user. This routine gets the X, Y, and other portions from the result of a particular fit object.

⟨*concordance*⟩=

```
cord.getdata <- function(object, newdata=NULL, cluster=NULL, need.wt, timefix=TRUE) {
    # For coxph object, don't reconstruct the model frame unless we must.
    # This will occur if weights, strata, or cluster are needed, or if
    #  there is a newdata argument.  Of course, if the model frame is
    #  already present, then use it!
    Terms <- terms(object)
    specials <- attr(Terms, "specials")
    if (!is.null(specials$tt))
        stop("cannot yet handle models with tt terms")

    if (!is.null(newdata)) {
        mf <- model.frame(object, data=newdata)
        y <- model.response(mf)
        if (!is.Surv(y)) {
            if (is.numeric(y) && is.vector(y))  y <- Surv(y)
            else stop("left hand side of the formula  must be a numeric vector or a survival obje
        }
        if (timefix) y <- aeqSurv(y)
        rval <- list(y= y, x= predict(object, newdata))
        # the type of prediction does not matter, as long as it is a
        #  monotone transform of the linear predictor
    }
    else {
        mf <- object$model
        y <- object$y
```

```
        if (is.null(y)) {
            if (is.null(mf)) mf <- model.frame(object)
            y <- model.response(mf)
        }
        if (!is.Surv(y)) {
            y <- Surv(y)
            if (timefix) y <- aeqSurv(y)
        }  # survival models will have already called timefix

        x <- object$linear.predictors    # used by most
        if (is.null(x)) x <- object$fitted.values # used by lm
        if (is.null(x)) {object$na.action <- NULL; x <- predict(object)}
        rval <- list(y = y, x= x)
    }

    if (need.wt) {
        if (is.null(mf)) mf <- model.frame(object)
        rval$weights <- model.weights(mf)
    }

    if (!is.null(specials$strata)) {
        if (is.null(mf)) mf <- model.frame(object)
        stemp <- untangle.specials(Terms, 'strata', 1)
        if (length(stemp$vars)==1) rval$strata <- mf[[stemp$vars]]
        else rval$strata <- strata(mf[,stemp$vars], shortlabel=TRUE)
    }

    if (is.null(cluster)) {
        if (!is.null(specials$cluster)) {
            if (is.null(mf)) mf <- model.frame(object)
            tempc <- untangle.specials(Terms, 'cluster', 1:10)
            ord <- attr(Terms, 'order')[tempc$terms]
            rval$cluster <- strata(mf[,tempc$vars], shortlabel=TRUE)
        }
        else if (!is.null(object$call$cluster)) {
            if (is.null(mf)) mf <- model.frame(object)
            rval$cluster <- model.extract(mf, "cluster")
        }
    }
    else rval$cluster <- cluster
    rval
}
```

The methods themselves, which are near clones of each other. There is one portion of these that is not very clear. I use the trick from nearly all calls to model.frame to deal with arguments that might be there or might not, such as newdata. Construct a call by hand by first subsetting

this call as Call[...], then replace the first element with the name of what I really want to call – quote(cord.work) –, add any other args I want, and finally execute it with eval(). The problem is that this doesn't work; the routine can't find cord.work since it is not an exported function. A simple call to cord.work is okay, since function calls inherit from the survival namespace, but cfun isn't a function call, it is an expression. There are 3 possible solutions

- bad: change eval(cfun, parent.frame()) to eval(cfun, evironment(coxph)), or any other function from the survival library which has namespace::survival as its environment. If the user calls concordance with ymax=zed, say, we might not be able to find 'zed'. Especially if they had called concordance from within a function. We need the call chain.
- okay: use cfun[[1]] ¡- cord.work, which makes a copy of the entire cord.work function and stuffs it in. The function isn't too long, so this is okay. If cord.work fails, the label on its error message won't be as nice since it won't have "cord.work" in it.
- speculative: make a function and invoke it. This creates a new function in the survival namespace, but evaluates it in the current context. Using parent.frame() is important so that I don't accidentally pick up 'nfit' say, if the user had used a variable of that name as one of their arguments.
  temp ¡- function()
  body(temp, environment(coxph)) ¡- cfun
  rval ¡- eval(temp(), parent.frame())

⟨*concordance*⟩=
```
 concordance.lm <- function(object, ..., newdata, cluster, ymin, ymax,
                            influence=0, ranks=FALSE, timefix=TRUE,
                            keepstrata=10) {
    Call <- match.call()
    fits <- list(object, ...)
    nfit <- length(fits)
    fname <- as.character(Call)  # like deparse(substitute()) but works for ...
    fname <- fname[1 + 1:nfit]
    notok <- sapply(fits, function(x) !inherits(x, "lm"))
    if (any(notok)) {
        # a common error is to mistype an arg, "ramk=TRUE" for instance,
        #  and it ends up in the ... list
        # try for a nice message in this case: the name of the arg if it
        #  has one other than "object", fname otherwise
        indx <- which(notok)
        id2 <- names(Call)[indx+1]
        temp <- ifelse(id2 %in% c("","object"), fname, id2)
        stop(temp, " argument is not an appropriate fit object")
    }

    cargs <- c("ymin", "ymax","influence", "ranks", "keepstrata")
    cfun <- Call[c(1, match(cargs, names(Call), nomatch=0))]
    cfun[[1]] <- cord.work   # or quote(survival:::cord.work)
```

```
    cfun$fname <- fname

    if (missing(newdata)) newdata <- NULL
    if (missing(cluster)) cluster <- NULL
    need.wt <- any(sapply(fits, function(x) !is.null(x$call$weights)))

    cfun$data <- lapply(fits, cord.getdata, newdata=newdata, cluster=cluster,
                        need.wt=need.wt, timefix=timefix)
    rval <- eval(cfun, parent.frame())
    rval$call <- Call
    rval
}

concordance.survreg <- function(object, ..., newdata, cluster, ymin, ymax,
                                timewt=c("n", "S", "S/G", "n/G", "n/G2", "I"),
                                influence=0, ranks=FALSE, timefix=FALSE,
                                keepstrata=10) {
    Call <- match.call()
    fits <- list(object, ...)
    nfit <- length(fits)
    fname <- as.character(Call)  # like deparse(substitute()) but works for ...
    fname <- fname[1 + 1:nfit]
    notok <- sapply(fits, function(x) !inherits(x, "survreg"))
    if (any(notok)) {
        # a common error is to mistype an arg, "ramk=TRUE" for instance,
        #  and it ends up in the ... list
        # try for a nice message in this case: the name of the arg if it
        #  has one other than "object", fname otherwise
        indx <- which(notok)
        id2 <- names(Call)[indx+1]
        temp <- ifelse(id2 %in% c("","object"), fname, id2)
        stop(temp, " argument is not an appropriate fit object")
    }

    cargs <- c("ymin", "ymax","influence", "ranks", "timewt", "keepstrata")
    cfun <- Call[c(1, match(cargs, names(Call), nomatch=0))]
    cfun[[1]] <- cord.work
    cfun$fname <- fname

    if (missing(newdata)) newdata <- NULL
    if (missing(cluster)) cluster <- NULL
    need.wt <- any(sapply(fits, function(x) !is.null(x$call$weights)))

    cfun$data <- lapply(fits, cord.getdata, newdata=newdata, cluster=cluster,
                        need.wt=need.wt, timefix=timefix)
    rval <- eval(cfun, parent.frame())
```

```
    rval$call <- Call
    rval
}

concordance.coxph <- function(object, ..., newdata, cluster, ymin, ymax,
                              timewt=c("n", "S", "S/G", "n/G", "n/G2", "I"),
                              influence=0, ranks=FALSE, timefix=FALSE,
                              keepstrata=10) {
    Call <- match.call()
    fits <- list(object, ...)
    nfit <- length(fits)
    fname <- as.character(Call)  # like deparse(substitute()) but works for ...
    fname <- fname[1 + 1:nfit]
    notok <- sapply(fits, function(x) !inherits(x, "coxph"))
    if (any(notok)) {
        # a common error is to mistype an arg, "ramk=TRUE" for instance,
        #  and it ends up in the ... list
        # try for a nice message in this case: the name of the arg if it
        #  has one other than "object", fname otherwise
        indx <- which(notok)
        id2 <- names(Call)[indx+1]
        temp <- ifelse(id2 %in% c("","object"), fname, id2)
        stop(temp, " argument is not an appropriate fit object")
    }

    # the cargs trick is a nice one, but it only copies over arguments that
    #  are present.  If 'ranks' was not specified, the default of FALSE is
    #  not set.  We keep it in the arg list only to match the documentation.
    cargs <- c("ymin", "ymax","influence", "ranks", "timewt", "keepstrata")
    cfun <- Call[c(1, match(cargs, names(Call), nomatch=0))]
    cfun[[1]] <- cord.work   # a copy of the function
    cfun$fname <- fname
    cfun$reverse <- TRUE

    if (missing(newdata)) newdata <- NULL
    if (missing(cluster)) cluster <- NULL
    need.wt <- any(sapply(fits, function(x) !is.null(x$call$weights)))

    cfun$data <- lapply(fits, cord.getdata, newdata=newdata, cluster=cluster,
                        need.wt=need.wt, timefix=timefix)
    rval <- eval(cfun, parent.frame())
    rval$call <- Call
    rval
}
```

The next routine does all of the actual work for a set of models. Note that because of the

call-through trick (fargs) exactly and only those arguments that are passed in are passed through to concordancefit. Default argument values for that function are found there. The default value for inflence found below is used in this routine, so it is important that they match.

⟨*concordance*⟩=
```
 cord.work <- function(data, timewt, ymin, ymax, influence=0, ranks=FALSE,
                       reverse, fname, keepstrata) {
     Call <- match.call()
     fargs <- c("timewt", "ymin", "ymax", "influence", "ranks", "reverse",
                "keepstrata")
     fcall <- Call[c(1, match(fargs, names(Call), nomatch=0))]
     fcall[[1L]] <- concordancefit

     nfit <- length(data)
     if (nfit==1) {
         dd <- data[[1]]
         fcall$y <- dd$y
         fcall$x <- dd$x
         fcall$strata <- dd$strata
         fcall$weights <- dd$weights
         fcall$cluster  <- dd$cluster
         rval <- eval(fcall, parent.frame())
     }
     else {
         # Check that all of the models used the same data set, in the same
         #  order, to the best of our abilities
         n <- length(data[[1]]$x)
         for (i in 2:nfit) {
             if (length(data[[i]]$x) != n)
                 stop("all models must have the same sample size")

             if (!identical(data[[1]]$y, data[[i]]$y))
                 warning("models do not have the same response vector")

             if (!identical(data[[1]]$weights, data[[i]]$weights))
                 stop("all models must have the same weight vector")
         }

         if (influence==2) fcall$influence <-3 else fcall$influence <- 1
         flist <- lapply(data, function(d) {
                         temp <- fcall
                         temp$y <- d$y
                         temp$x <- d$x
                         temp$strata <- d$strata
                         temp$weights <- d$weights
                         temp$cluster <- d$cluster
```

```
                    eval(temp, parent.frame())
                })

        for (i in 2:nfit) {
            if (length(flist[[1]]$dfbeta) != length(flist[[i]]$dfbeta))
                stop("models must have identical clustering")
        }
        count = do.call(rbind, lapply(flist, function(x) {
            if (is.matrix(x$count)) colSums(x$count) else x$count}))

        concordance <- sapply(flist, function(x) x$concordance)
        dfbeta <- sapply(flist, function(x) x$dfbeta)

        names(concordance) <- fname
        rownames(count) <- fname

        wt <- data[[1]]$weights
        if (is.null(wt)) vmat <- crossprod(dfbeta)
        else vmat <- t(wt * dfbeta) %*% dfbeta
        rval <- list(concordance=concordance, count=count,
                     n=flist[[1]]$n, var=vmat,
                     cvar= sapply(flist, function(x) x$cvar))

        if (influence==1) rval$dfbeta <- dfbeta
        else if (influence ==2) {
            temp <- unlist(lapply(flist, function(x) x$influence))
            rval$influence <- array(temp,
                                    dim=c(dim(flist[[1]]$influence), nfit))
        }

        if (ranks) {
            temp <- lapply(flist, function(x) x$ranks)
            rdat <- data.frame(fit= rep(fname, sapply(temp, nrow)),
                               do.call(rbind, temp))
            row.names(rdat) <- NULL
            rval$ranks <- rdat
        }
     }

    class(rval) <- "concordance"
    rval
}
```

Last, a few miscellaneous methods

⟨*concordance*⟩=

```
coef.concordance <- function(object, ...) object$concordance
```