Survival Package Functions

Terry Therneau

January 6, 2023

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. 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 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.

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), ...) {

    missing.ties <- missing(ties) & missing(method) #see later multistate sect
    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 <- "cluster"
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)

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

if (length(tcl) > 0) { # there is one
    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")

    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")

    # [.terms is broken at least through R 4.1; use our
    #  local drop.special() function instead.
    Terms <- drop.special(Terms, tcl)
    formula <- Call$formula <- formula(Terms)
}

# 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)

# 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) {
    if (length(attr(Terms, "specials")$frailty) >0)
        stop("multi-state models do not currently support frailty terms")
    if (length(attr(Terms, "specials")$pspline) >0)
        stop("multi-state models do not currently support pspline terms")
    if (length(attr(Terms, "specials")$ridge) >0)
        stop("multi-state models do not currently support ridge penalties")
    if (!missing.ties) method <- ties <- "breslow"
}

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 Surv2
    # begin tt() preprocessing
    ⟨coxph-transform⟩
    # end tt() preprocessing
    }

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) {
    # tmap starts with a "(Baseline)" row, which we want
    # strats is indexed off the data frame, which includes the response, so
    #  turns out to be correct for the remaining rows of tmap
    smap <- tmap[c(1L, strats),]
    smap[-1,] <- ifelse(smap[-1,] >0, 1L, 0L)
    if (nrow(smap) > 2) {
        # multi state with more than 1 strata statement -- really unusual
        temp <- smap[-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 smap <- 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 `smap`. 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, smap, as.integer(istate), X, Y, strata=istrat,
                  states=states)

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

Xsave <- X  # the originals may be needed later
Ysave <- Y
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")
id <- model.extract(mf, "id")   # update the id and/or cluster, if present
cluster <- model.extract(mf, "cluster")

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 to ties", 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)  {
        if (multi) fit$x <- Xsave else fit$x <- X
        if (length(timetrans)) fit$strata <- istrat
        else if (length(strats)) fit$strata <- strata.keep
    }
    if (y)  {
        if (multi) fit$y <- Ysave else fit$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.

A multi-state model will have a matrix of linear predictors and of residuals. Each has a column for each transition and a row for each subject. The rows are with respect to the starting X and Y, not the expanded ones which were used to compute the coefficients. The expanded linear predictor is easy: Xbeta where beta is the matrix form of the coefficients. Residuals are a bit more nuisance: if an observation was a risk for an a:b transition, it will appear in the a:b strata of the expanded X matrix, and that residual fills in the appropriate row/col. If it was not at risk for said transition, the residual is zero. There is, however, a further problem. Any transitions for which there are no covariates were not sent across as a strata to the fitting routine — they would create a stratum where all covariates = 0, which cause computation for no cause. But that also means that the martingale residuals are not computed for those rows of the data.

⟨*coxph-finish*⟩=

```
 if (!is.null(weights) && any(weights!=1)) fit$weights <- weights
 if (multi) {
     fit$transitions <- transitions
     fit$states <- states
     fit$cmap <- cmap
     fit$smap <- smap   # why not 'stratamap'?  Confusion with fit$strata
     nonzero <- which(colSums(cmap)!=0)
     fit$rmap <- cbind(row=xstack$rindex, transition= nonzero[xstack$transition])

     # 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]))
     newname <- paste0(names(fit$coefficients), suffix)
     if (any(covlist2$phbaseline > 0)) {
         # for proportional baselines, use a better name
         base  <- colnames(tmap)[covlist2$phbaseline]
         child <- colnames(tmap)[which(covlist2$phbaseline >0)]
         indx <- 1 + length(newname) - length(base):1 # coefs are the last ones
         newname[indx] <-  paste0("ph(", child, "/", base, ")")
         phrow <- apply(cmap, 1, function(x) all(x[x>0] %in% indx))
         matcoef <- cmap[!phrow,,drop=FALSE ] # ph() terms exluded
         }
     else matcoef <- cmap
     names(fit$coefficients) <- newname

     if (FALSE) {
         # an idea that was tried, then paused: make the linear predictors
         # and residuals into matrices with one column per transition
         matcoef[matcoef>0] <- fit$coefficients[matcoef]
         temp <- Xsave %*% matcoef
         colnames(temp) <- colnames(cmap)
```

```
        fit$linear.predictors <- temp

        temp <- matrix(0., nrow=nrow(Xsave), ncol=ncol(fit$cmap))
        temp[cbind(xstack$rindex, xstack$transition)] <- fit$residuals
        # if there are any transitions with no covariates, residuals have not
        #  yet been calculated for those.
        if (any(colSums(cmap) ==0)) {
            from.state <- as.numeric(sub(":.*$", "", colnames(cmap)))
            to.state   <- as.numeric(sub("^.*:", "", colnames(cmap)))
           # warning("no covariate residuals not filled in")
        }
        fit$residuals <- temp
    }
    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

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

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

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

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*⟩=

```
#define NOTDONE -1.1

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 == NOTDONE) {  /* 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 identical 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.

Later update: User feedback about an "infinite computation" proved that the case most definitely does exist: in one strata their first 65 rows had x=0 for one of the variables. Not actually infinite compute time, but close enough. One solution is to pick a value that will never occur as the first derivative. That is impossible, but actually anything other than 0 should never be the first derivative for more than a single (d,n) combination. We use a negative number for the constant NOTDONE since d0 must be positive, and thus no issues arise there.

⟨*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] == NOTDONE) { /* 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] == NOTDONE) { /*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, notfinite;

    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 shortest 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;   /* first obs of current stratum */
 ndeath=0; nrisk=0;
 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;  /* 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++ = NOTDONE;
        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 || isfinite(loglik[0])==0) { /* give up on overflow */
    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);

    notfinite = 0;
    for (i=0; i<nvar; i++) {
            if (isfinite(u[i]) ==0) notfinite=2;     /* infinite score stat */
        for (j=0; j<nvar; j++) {
            if (isfinite(imat[i][j]) ==0) notfinite =3; /*infinite imat */
            }
        }
    if (isfinite(newlk) ==0) notfinite =4;

    if (notfinite==0 && 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 (notfinite > 0 || 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 */


/*
** We end up here only if we ran out of iterations
**  recompute the last good version of the loglik and imat
** If maxiter =0 or 1, though, leave well enough alone.
```

```
 */
 if (maxiter > 1) {
    for (i=0; i< nvar; i++) beta[i] = oldbeta[i];
    newlk =0;
    for (i=0; i<nvar; i++) {
        u[i] =0;
        for (j=0; j<nvar; j++)
            imat[i][j] =0;
    }
    ⟨excox-addup⟩
 }
 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$weights  # let offsets/weights be NULL until needed
    offset <- NULL
    offset.mean <- 0
    X <- object[['x']]
}
else {
    weights <- model.weights(mf)
    offset <- model.offset(mf)
    if (is.null(offset)) offset.mean <- 0
    else {
        if (is.null(weights)) offset.mean <- mean(offset)
        else offset.mean <- sum(offset * (weights/sum(weights)))
    }
    X <- model.matrix.coxph(object, data=mf)
    if (is.null(Y) || coxms) {
        Y <- model.response(mf)
        if (is.null(object$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)  # removed 2/2022
     }
     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$means) ==0) { # a model with only an offset term
     # Give it a dummy X so the rest of the code goes through
     #  (This case is really rare)
     # se.fit <- FALSE
     X <- matrix(0., nrow=n, ncol=1)
     if (is.null(offset)) offset <- rep(0, n)
     xcenter <- offset.mean
     coef <- 0.0
     varmat <- matrix(0.0, 1, 1)
     risk <- rep(exp(offset- offset.mean), length=n)
 }
 else {
     varmat <- object$var
     beta <- ifelse(is.na(object$coefficients), 0, object$coefficients)
     xcenter <- sum(object$means * beta) + offset.mean
     if (!is.null(object$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

     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)
         y2 <- NULL  # a dummy to carry along, for the call to coxsurv.fit
     }
     ⟨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 <- 0
    # a model with only an offset, but newdata containing a value for it
    if (length(object$means)==0) x2 <- 0
    else 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) {
             ⟨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⟩=
 if (is.matrix(result$surv)) nr <- nrow(result$surv)
 else nr <- length(result$surv)   # if newdata had only one row
 indx1 <- split(1:nr, rep(1:length(result$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

 if (is.matrix(result$surv)) {
     for(i in 2:length(rows)) rows[[i]] <- rows[[i]]+ (i-1)*nr #linear subscript
     indx4 <- unlist(rows)   #index for surv and std.err
 } else indx4 <- indx2
 temp <- result$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)
    }

if (se.fit) {
    result$logse = TRUE   # this will migrate to solutio
    # In this particular case, logse=T and they are the same
    #  Other cases await addition of code
    if (stype==2) result$std.chaz <- result$std.err
}

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

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

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_iVd_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 ≥ 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(int   *ndeath,   double *risk,    double *wt,
             int   *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/x12̂`

  xbar `xsum/x12̂`
- d=2

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

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

  xbar (1/2) `(xsum/x12̂ + (xsum - 1/2 x3)/(x1- x2/2)2̂)`
- 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/x12̂ + (xsum - 1/3 xsum2)/(x1- x2/3)2̂ +`
  `(xsum - 2/3 xsum2)/(x1- 2/3 x3)2̂)`
- 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(int *n2,      int *nvar2,   int *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=FALSE, 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$smap["(Baseline)",]
    baselinecoef <- rbind(temp, coef= 1.0)
    if (any(duplicated(temp))) {
        # We have shared hazards
        # If there are k duplicates, then the last k coefficient in beta are
        #  the coefs that match them
        idup <- duplicated(temp)
        ncoef <- length(object$coefficients)
        ndup <- sum(idup)
        # shared baseline coefficients are last in the coefficient vector
        i <- seq(to = ncoef, length=ndup)
        baselinecoef[2, idup] <- exp(object$coefficients[i])

        # which rows of cmap point to scale coefs?
        phbase <- apply(object$cmap, 1, function(i) any(i > ncoef-ndup))
    }
    else phbase <- rep(FALSE, nrow(object$cmap))

    # process options, set up Y and the model frame for the original data
     ⟨survfit.coxph-setup1⟩
     ⟨survfit.coxph-setup2⟩
    istate <- model.extract(mf, "istate")

    #deal with start time, by throwing out observations that end before then
    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, as a double check
# that the data set hasn't been modified
mcheck <- survcheck2(Y, oldid, istate)
transitions <- mcheck$transitions
if (!identical(object$states, mcheck$states))
    stop("failed to rebuild the data set")
if (is.null(istate)) istate <- mcheck$istate
else {
    # if istate has unused levels, mcheck$istate won't have them so they
    #  need to be dropped.
    istate <- factor(istate, object$states)
    # a new level in state should only happen if someone has mucked up the
    #  data set used in the coxph fit
    if (any(is.na(istate))) stop("unrecognized initial state, data changed?")
}

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

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
# Note the dropzero=FALSE argument: if there is a transition with no
#  covariates we still need it expanded; this differs from coxph.
# A second differnence is tstrata: force stacker to think that every
#  transition is a unique hazard, so that it does proper expansion.
cluster <- model.extract(mf, "cluster")
tstrata <- object$smap
tstrata[1,] <- 1:ncol(tstrata)
xstack <- stacker(object$cmap, tstrata, as.integer(istate), X, Y,
                  as.integer(strata),
                  states= object$states, dropzero=FALSE)
if (length(position) >0)
    position <- position[xstack$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
```

Fix up the X matrix to avoid huge values. In the single state case this is fairly straightforward: use $(X - 1m')\beta = X\beta - m'\beta$ where m is the vector of centering constants found in the `object$means` component. However, in multi-state there will often be covariates that are part of one transition but not another, and if one of them is wild we will want different centering for each transition. (Not yet implemented).

⟨*survfit.coxph-setup2d*⟩=
```
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 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
# temp will be a matrix of coefficients, with ncol = number of transtions
#  and nrow = the covariate set of a "normal" Cox model.
# x2 will have one row per desired curve and one col per 'normal' covariate.
risk2 <- exp(x2 %*% ifelse(is.na(temp), 0, temp) - xcenter)
# risk2 has a risk score with rows= curve and cols= transition
```

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 * baseline hazard combination.
- `baselinecoef` maps from baseline hazards to transitions. It has one column per transition, which baseline 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$states
nstate <- length(states)
from <- as.numeric(sub(":.*$", "", colnames(object$smap)))
to   <- as.numeric(sub("^.*:", "", colnames(object$smap)))
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 in user printout, nothing else. (That part has actually already been done via the survfitCI call.)
- x2 and risk2 are the covariates and risk scores for the predicted values. These do not involve any ph(a:b) coefficients.
- baselinecoef encodes shared hazards
- hfill control mapping from fitted 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

The cn matrix below contains all the subtotals we need. Say that transitions 4, 5, and 6 have a shared hazard, with bcoef[2,] values of 1, 1.3, .4 (the first coef is always 1). Then the underlying hazard will base = (events[3] + events[4] + events[5])/ (nrisk[3] + 1.3* nrisk[4] + .4*nrisk[5]), and the 3 individual hazards are 1*base, 1.3*base and .4*base. If there are no shared hazards this can be computed more simply of course.

```
⟨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) {
    ny <- ncol(y)
    sort2 <- order(istrat, y[,ny-1L]) -1L
    ntime <- length(utime)
    storage.mode(weight) <- "double"  #failsafe

    # this returns all of the counts we might desire.
    if (ny ==2) {
        fit <- .Call(Ccoxsurv1, utime, y, weight, sort2, istrat, x, risk)
        cn <- fit$count
        dim(cn) <- c(length(utime), fit$ntrans, 10)
    }
```

```
else {
    sort1 <- order(istrat, y[,1]) -1L
    fit <- .Call(Ccoxsurv2, utime, y, weight, sort1, sort2, position,
                    istrat, x, risk)
    cn <- fit$count
    dim(cn) <- c(length(utime), fit$ntrans, 12)
}
# cn is returned as a matrix since there is an allocMatrix C macro, but
#  no allocArray macro.  So we first reset the dimensions.
# The first dimension is time
# Second is the transition, same order as columns of bcoef
# Third is the count type: 1-3 = at risk (unweighted, with case weights,
#  with casewt * risk wt), 4-6 = events (unweighted, case, risk),
#  7-8 = censored events, 9-10 = censored, 11-12 = Efron

# We will use events/(at risk) = cn[,,5]/cn[,,3] a few lines below; avoid 0/0
# If there is no one at risk there are no events, obviously.
# cn[,,1] is the safer check since it is an integer, but if there are weights
#  and a subject with weight=0 were the only one at risk, we need cn[,,2]
# (Users should never use weights of 0, but someone, somewhere, will do it.)
none.atrisk <- (cn[,,1]==0 | cn[,,2]==0)
if (ctype ==1) {
    denom1 <- ifelse(none.atrisk, 1, cn[,,3])   # avoid a later 0/0
    denom2 <- ifelse(none.atrisk, 1, cn[,,3]^2)
} else {
    denom1 <- ifelse(none.atrisk, 1, cn[,,9])
    denom2 <- ifelse(none.atrisk, 1, cn[,,10])
}

# We want to avoid 0/0. If there is no one at risk (denominator) then
# by definition there will be no events (numerator), and that element of
# the hazard is by definintion also 0.
if (any(duplicated(bcoef[1,]))) {
    # there are shared hazards: we have to collapse and then expand
    if (all(bcoef[1,] == bcoef[1,1])) design <- matrix(1, nrow=ncol(bcoef))
    else design <- model.matrix(~factor(zed) -1, data.frame(zed=bcoef[1,]))
    colnames(design) <- 1:ncol(design)  # easier to read when debuggin
    events <- cn[,,5] %*% design
    if (ctype==1) atrisk <- cn[,,3]  %*% design
    else          atrisk <- cn[,,9] %*% design
    basehaz <- events/ifelse(atrisk<=0, 1, atrisk)
    hazard <- basehaz[,bcoef[1,]] * rep(bcoef[2,], each=nrow(basehaz))
}
else {
    if (ctype==1) hazard <- cn[,,5]/ifelse(cn[,,3]<=0, 1, cn[,,3])
    else          hazard <- cn[,,5]/ifelse(cn[,,9] <=0, 1, cn[,,9])
}

# Expand the result, one "hazard set" for each row of x2
nx2 <- nrow(x2)
h2 <- array(0, dim=c(nrow(hazard), nx2, ncol(hazard)))
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)    # dummy H to drive esetup
    diag(H) <- diag(H) -rowSums(H)
    esetup <- survexpmsetup(H)
}

for (i in 1:nx2) {
    h2[,i,] <- apply(hazard * rep(risk2[i,], each=ntime), 2, cumsum)
    if (FALSE) {  # if (se.fit) eventually
        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) {
        if (any(hazard[j,] > 0)) { # hazard =0 for censoring times
            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) ⟶ state 1
- (not yet in state 2) ⟶ 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.

Setup 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')$cluster)) {
    temp <- untangle.specials(Terms, 'cluster', 1)
    Terms  <- drop.special(Terms, attr(Terms, "specials")$cluster)
    }

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") ||
    (reference=="zero" && any(object[["means"]] !=0))) {
    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$strata
            else oldstrat <- rep(0L, n)
            weights <-  rep(1.,n)
            offset <-   0
        }
    }
    else {
        # I won't need strata in this case either
        if (has.strata) {
            Terms2  <- drop.special(Terms2, attr(Terms2, "specials")$strata)
            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$vars], shortlabel=TRUE)
            if (any(is.na(match(levels(newstrat), levels(oldstrat)))))
                stop("New data has a strata not found in the original model")
            else newstrat <- factor(newstrat, levels=levels(oldstrat)) #give it all
            if (length(strat.term$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.
- 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. The timewt option has no effect for other than survival data.

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/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)
        if (ncol(Y) == 3 && timewt %in% c("S/G", "n/G", "n/G2"))
            stop(timewt, " timewt option not supported for (time1, time2) data")
    } 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 if (is.logical(Y)) Y <- as.numeric(Y)
        else stop("left hand side of the formula must be a numeric vector,
 survival object, or an orderable factor")
        timewt <- "n"
        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 (is.null(x$var)) {
        # Result of a call with std.err = FALSE
        cat("Concordance= ", format(x$concordance, digits=digits), "\n")
    } else {
        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.

It turns out that for moderate sized data sets, the computation of the survival curve S (or censoring G) takes as much or more compuational time as the routine proper. For this reason, make sure not to call survfit unless we have to. Since we are calling those routines for data with no grouping variable, no standard error, and none of the other options, speed could be gained by moving the computation of those quantities inside of the C code.

```
⟨concordancefit⟩=
concordancefit <- function(y, x, strata, weights, ymin=NULL, ymax=NULL,
                           timewt=c("n", "S", "S/G", "n/G2", "I"),
                           cluster, influence=0, ranks=FALSE, reverse=FALSE,
                           timefix=TRUE, keepstrata=10, std.err =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 (!is.null(ymin) && !is.numeric(ymin)) stop("ymin must be numeric")
    if (!is.null(ymax) && !is.numeric(ymax))    stop("ymax must be numeric")
    if (!std.err) {ranks <- FALSE; influence <- 0;}

    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")
        storage.mode(weights) <- "double" # for the .Call, in case of integers
    }
    if (is.Surv(y)) {
        ny <- ncol(y)
        if (ny == 3 && timewt %in% c("S/G", "n/G2"))
            stop(timewt, " timewt option not supported for (time1, time2) data")
        if (!is.null(attr(y, "states")))
            stop("concordance not defined for a multi-state outcome")
        if (timefix) y <- aeqSurv(y)
        if (!is.null(ymin)) {
            censored <- (y[,ny] ==0)
            if (any(y[censored, ny-1] < ymin))
                stop("data has a censored value less than ymin")
            else y[,ny-1] <- pmax(y[,ny-1], ymin)
        }
    } else {
        # should only occur if another package calls this routine
        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")
    }

    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")
storage.mode(y) <- "double"   # in case of integers, for the .Call

if (!is.null(ymin) && any(y[, ny-1] < ymin))
    y[,ny-1] <- pmax(y[,ny-1], ymin)
# ymax is dealt with in the docount routine, as shifting end of a (t1, t2)
#  interval could generate invalid data

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’) {
    n <- length(risk)
    ny <- ncol(y)   # 2 or 3

    if (sum(y[,ncol(y)]) ==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))
    }

    # 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[,ny]) <2) timeopt <- 'n'

    # order the data: reverse time, censors before deaths
    if (ny ==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
    }

    if (timeopt == 'n') {
        deaths <- y[,ny] > 0
        etime  <- sort(unique(y[deaths, ny-1]))  # event times
    }
    else {
        if (ny==2) {
            sort.stop <- order(-y[,1], y[,2], risk) -1L
            gfit <- .Call(Cfastkm1, y, wts, sort.stop)
        } else {
            sort.stop  <- order(-y[,2], y[,3], risk) -1L   #order by endpoint
            sort.start <- order(-y[,1]) -1L
            gfit <- .Call(Cfastkm2, y, wts, sort.stop, sort.start)
        }
        etime <- gfit$etime
    }

    timewt <- switch(timeopt,
                     "S"   = sum(wts)* gfit$S/gfit$nrisk,
                     "S/G" = sum(wts)* gfit$S/ (gfit$G * gfit$nrisk),
                     "n" =   rep(1.0, length(etime)),
                     "n/G2"= 1/gfit$G^2,
                     "I"  =  1/gfit$nrisk)
    if (any(!is.finite(timewt))) stop("program error, notify author")

    if (!is.null(ymax)) timewt[etime > ymax] <- 0

    # 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]

    if (std.err) {
        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")
            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)
    count2 <- fit$count[1:5]
    vcox <- fit$count[6]
    fit$count <- fit$count[1:5]
    if (std.err) 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)
    })
    temp <-  t(sapply(tfit, function(x) x$count))
    fit <- list(count = temp[,1:5])
    count2 <- colSums(fit$count)
    if (!keepstrata) fit$count <- count2
    vcox <- sum(temp[,6])
    if (std.err) {
        imat <- do.call("rbind", lapply(tfit, function(x) x$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$count)
somer <- (count2[1] - count2[2])/npair
if (std.err) {
    dfbeta <- weights*((imat[,1]- imat[,2])/npair -
                      (somer/npair)* rowSums(imat[,1:3]))
    if (!missing(cluster) && length(cluster)>0) {
        dfbeta <- tapply(dfbeta, cluster, sum)
        dfbeta <- ifelse(is.na(dfbeta),0, dfbeta)  # if cluster is a factor
    }
    var.somer <- sum(dfbeta^2)
    rval <- list(concordance = (somer+1)/2, count=fit$count, n=n,
                 var = var.somer/4, cvar=vcox/(4*npair^2))
    }
else  rval <- list(concordance = (somer+1)/2, count=fit$count, n=n)

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(Terms, data=newdata)
        y <- model.response(mf)
        # why not model.frame(object, newdata)?  Because model.frame.coxph
        #  uses the Call to fill in names for subset, if present, which of
        #  course is not relevant to the new data. model.frame.lm does the
        #  same, btw.
        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 objec
        }
        if (timefix) y <- aeqSurv(y)

        # special handling for coxph objects: object to tt()
        specials <- attr(Terms, "specials")
        if (!is.null(specials$tt)) stop("newdata not allowed for tt() terms")


        yhat <- predict(object, newdata=newdata, na.action = na.omit)
        # yes, the above will construct mf again, but all the special processing
        #  we get for lm, glm, coxph  is worth it.
        rval <- list(y= y, x= as.vector(yhat))

        # recreate the strata, if needed
        if (length(attr(Terms, "specials")$strata) > 0) {
            stemp <- untangle.specials(Terms, ’strata’, 1)
            if (length(stemp$vars)==1) strata.keep <- mf[[stemp$vars]]
            else strata.keep <- strata(mf[,stemp$vars], shortlabel=TRUE)
            rval$strata <- as.integer(strata.keep)
        }
    }
    else {
        mf <- object$model
        y <- object$y
        if (is.null(y)) {
            if (is.null(mf)) mf <- model.frame(object)
            y <- model.response(mf)
        }
        x <- object$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 (!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 (need.wt) {
        if (is.null(mf)) mf <- model.frame(object)
        rval$weights <- model.weights(mf)
    }

    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/G2", "I"),
                                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, "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/G2", "I"),
                              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, "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$concordance
 vcov.concordance <- function(object, ...) object$var
```

The C routine returns an influence matrix with one row per subject i, and columns giving the partial with respect to w_i for the number of concordant, discordant, tied on x and ties on y pairs. Somers' d is $(C-D)/m$ where $m = C+D+T$ is the total number of comparable pairs, which does not count the tied-on-y column. For any given subject or cluster k (for grouped jackknife) the IJ estimate of the variance is

$$
\begin{aligned}
V & \sum_k \left(\frac{\partial d}{\partial w_k}\right)^2 \\
\frac{\partial d}{\partial w_k} &= \frac{1}{m}\left[\frac{\partial C-D}{\partial w_k} - d\frac{\partial C+D+T}{\partial w_k}\right]
\end{aligned}
$$

The C code looks a lot like a Cox model: walk forward through time, keep track of the risk sets, and add something to the totals at each death. What needs to be summed is the rank of the event subject's x value, as compared to the value for all others at risk at this time point. For notational simplicity let $Y_j(t_i)$ be an indicator that subject j is at risk at event time t_i, and $Y^*_j(t_i)$ the more restrictive one that subject j is both at risk and not a tied event time. The values we want at time t_i are

$$C_i = v_i \delta_i w_i \sum_j w_j Y_j^*(t_i) \left[I(x_i < x_j) \right] \tag{8}$$

$$D_i = v_i \delta_i w_i \sum_j w_j Y_j^*(t_i) \left[I(x_i > x_j) \right] \tag{9}$$

$$T_i = v_i \delta_i w_i \sum_j w_j Y_j^*(t_i) \left[I(x_i = x_j) \right] \tag{10}$$

$$\tag{11}$$

In the above v is an optional time weight, which we will discuss later. The normal concordance definition has $v = 1$. C, D, and T are the number of concordant, discordant, and tied pairs, respectively, and $m = C + D + T$ will be the total number of concordant pairs. Somers' d is $(C - D)/m$ and the concordance is $(d + 1)/2 = (C + T/2)/m$.

The primary compuational question is how to do this efficiently, i.e., better than a naive algorithm that loops across all $n(n - 1)/2$ possible pairs. There are two key ideas.

1. Rearrange the counting so that we do it by death times. For each death we count the number of other subjects in the risk set whose score is higher, lower, or tied and add it into the totals. This neatly solves the question of time-dependent covariates.

2. Counting the number with higher, lower, and tied x can be done in $O(\log_2 n)$ time if the x data is kept in a binary tree.

Figure 1 shows a balanced binary tree containing 13 risk scores. For each node the left child and all its descendants have a smaller value than the parent, the right child and all its descendents have a larger value. Each node in figure 1 is also annotated with the total weight of observations in that node and the weight for itself plus all its children (not shown on graph). Assume that the tree shown represents all of the subjects still alive at the time a particular subject "Smith" expires, and that Smith has the risk score of 19 in the tree. The concordant pairs are those with a risk score > 19, i.e., both $\hat{y} = x$ and y are larger, discordant are < 19, and we have no ties. The totals can be found by

1. Initialize the counts for discordant, concordant and tied to the values from the left children, right children, and ties at this node, respectively, which will be $(C, D, T) = (1, 1, 0)$.

2. Walk up the tree, and at each step add the (parent + left child) or (parent + right child) to either D or C, depending on what part of the tree has not yet been totaled. At the next node (8) $D = D + 4$, and at the top node $C = C + 6$.

There are 5 concordant and 7 discordant pairs. This takes a little less than $\log_2(n)$ steps on average, as compared to an average of $n/2$ for the naive method. The difference can matter when n is large since this traversal must be done for each event.

The classic way to store trees is as a linked list. There are several algorithms for adding and subtracting nodes from a tree while maintaining the balance (red-black trees, AA trees, etc) but we take a different approach. Since we need to deal with case weights in the model and we know all the risk score at the outset, the full set of risk scores is organised into a tree at the beginning,

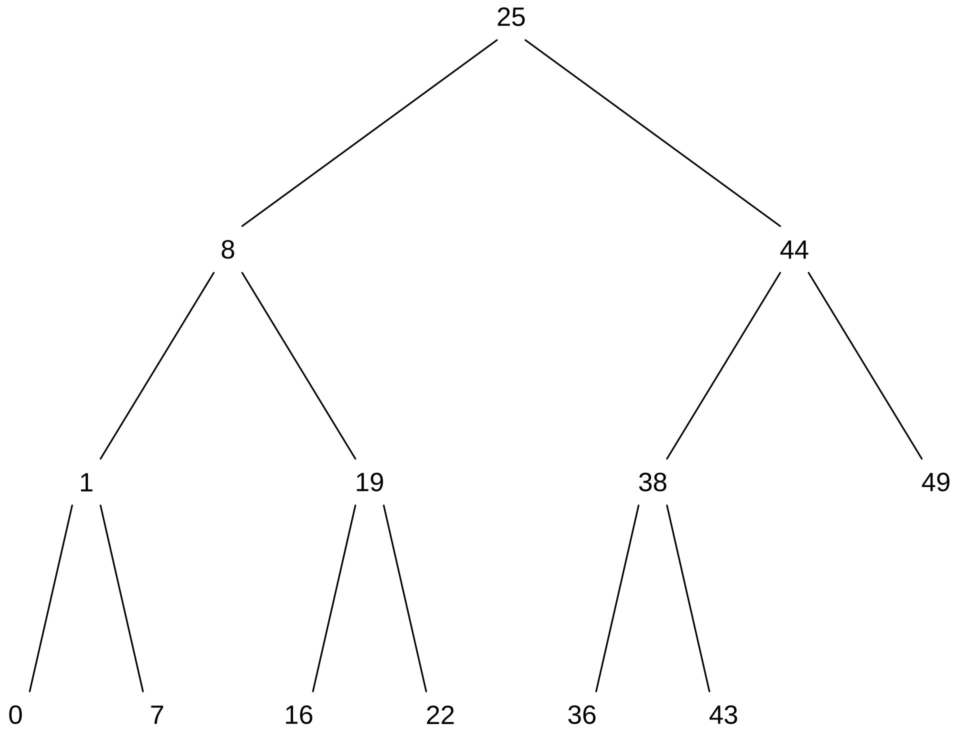

Figure 1: A balanced tree of 13 nodes.

updating the sums of weights at each node as observations are added or removed from the risk set.

If we internally index the nodes of the tree as 1 for the top, 2–3 for the next horizontal row, 4–7 for the next, …then the parent-child traversal becomes particularly easy. The parent of node i is $i/2$ (integer arithmetic) and the children of node i are $2i$ and $2i+1$. In C code the indices start at 0 of course. The following bit of code arranges data into such a tree.

```
⟨btree⟩=
 btree <- function(n) {
    tfun <- function(n, id, power) {
        if (n==1L) id
        else if (n==2L) c(2L *id + 1L, id)
        else if (n==3L) c(2L*id + 1L, id, 2L*id +2L)
        else {
            nleft <- if (n== power*2L) power  else min(power-1L, n-power%/%2L)
            c(tfun(nleft, 2L *id + 1L, power%/%2), id,
              tfun(n-(nleft+1L), 2L*id +2L, power%/%2))
        }
    }
    tfun(as.integer(n), 0L, as.integer(2^(floor(logb(n-1,2)))))
 }
```

Referring again to figure 1, `btree(13)` yields the vector `7 3 8 1 9 4 10 0 11 5 12 2 6` meaning that the smallest element will be in position 8 of the tree, the next smallest in position 4, etc, and using indexing that starts at 0 since the results will be passed to a C routine. The code just above takes care to do all arithmetic as integer. This actually made almost no difference in the compute time, but it was an interesting exercise to find that out.

The next question is how to compute a variance for the result. One approach is to compute an infinitesimal jackknife (IJ) estimate, for which we need derivatives with respect to the weights. Looking back at equation (8) we have

$$C = \sum_i w_i \delta_i \sum_j Y_j^*(t_i) w_j I(x_i < x_j) \tag{12}$$

A given subject's weight appears multiple times, once when they are an event ($w_i \delta_i$), and then as part of the risk set for other's events. I avoided this for some time because it looked like an $O(nd)$ process to separately update each subject's influence for each risk set they inhabit, but David Watson pointed out a path forward. The solution is to keep two trees. Tree 1 contains all of the subjects at risk. We traverse it when each subject is added in, updating the tree, and traverse it again at each death, pulling off values to update our sums. The second tree holds only the deaths and is updated at each death; it is read out twice per subject, once just after they enter the risk set and once when they leave.

The basic algorithm is to move through an outer and inner loop. The outer loop moves across unique times, the inner for all obs that share a death time. We progress from largest to smallest time. Dealing with tied deaths is a bit subtle.

- All of the tied deaths need to be added to the event tree before subtracting the tree values from the "initial" influence matrix, since none of the tied subjects are in the comparison set for each other.
- Changes to the overall concordance/discordance counts need to be done for all the ties before adding them into the main tree, for the same reason.
- The Cox model variance just below has to be added up sequentially, one term after each addition to the main tree.

Thus the inner loop must be repeated at least twice.

A second variance computation treats the data as a Cox model. Create zero-centered scores for all subjects in the risk set:

$$\begin{aligned} z_i(t) &= \sum_{j \in R(t)} w_j \text{sign}(x_i - x_j) \\ D - C &= \sum_i \delta_i z_i(t_i) \end{aligned} \tag{13}$$

At any event time $\sum w_i z_i = 0$. Equation (13) is the score equation for a Cox model with time-dependent covariate z. When two subjects have an event at the same time, this formulation treats each of them as being in the other's risk set whereas the concordance treats them as incomparable — how can they be the same? The trick is that $D - C$ does not change: the tied pairs add equally to D and C. Under the null hypothesis that the risk score is not related to outcome, each term in (13) is a random selection from the z scores in the risk set, and the variance of the addition is the variance of z, the sum of these over deaths is the Cox model information matrix, which is also the variance of the score statistic. The mean of z is always zero, so we need to keep track of $\sum w_i z^2$.

How can we do this efficiently? First note that z_i can be written as sum(weights for smaller x) - sum(weights for larger x), and in fact the weighted mean for any slice of x, $a < x < b$, is exactly the same: mean = sum(weights for x values below the range) - sum(weights above the range). The second trick is to use an ANOVA decomposition of the variance of z into within-slice and between-slice sums of squares, where the 3 slices are the z scores at a given x value (node of the tree), weights for score below that cutpoint, and above. Assume that a new observation k has just been added to the tree. This will add w_k to all the z values above, and to the weighted mean of all those above, $-w_k$ to the values and means below, and 0 to the values and means of any tied observations. Thus none of the current 'within' SS change. Let s_a, s_b and s_0 be the current sum of weights above, below, and at the node of the tree. The mean for the above group was $(s_b + s_0)$ with between SS contribution of $s_a(s_b + s_0)^2$. The below mean was $-(s_a + s_0)$ with between SS contribution of $s_b(s_a + s_0)^2$. The change to the between SS from adding the new subject is

$$s_a\left((s_b + s_0 + w_k)^2 - (s_b + s_0)^2\right) = s_a(2w_k(s_b + s_0) + w_k^2)$$

while the change in between SS for the below group is $s_b(2w_k(s_a + s_0) + w_k^2)$, and there is no change for the prior observations in the middle group. Last we add $w_k z_k^2 = w_k(s_b - s_a)^2$ to the sum for the new observation. Putting all this together the change is

$$w_k\left(s_a(w_k + (s_b + s_c)) + s_b(w_k + (s_a + s_c)) + (s_a - s_b)^2\right)$$

We can now define the C-routine that does the bulk of the work. First we give the outline shell of the code and then discuss the parts one by one. This routine is for ordinary survival data, and will be called once per stratum. Input variables are

n the number of observations

y matrix containing the time and status, data is sorted by descending time, with censorings precedint deaths.

x the tree node at which this observation's risk score resides

wt case weight for the observation

The routine will return list with three components:

- count, a vector containing the weighted number of concordant, discordant, tied on x but not y, and tied on y pairs. The weight for a pair is $w_i w_j$.
- resid, a three column matrix with one row per event, containing the score residual at that event, the time weight, and the case weight of the event. The residual is $s_i - \overline{s}$: the percentile of the subject with the event, among all those at risk, minus the mean. This is the "Cox model" way of looking at that data. A weighted sum of the residuals will equal C-D = concordant - discordant.
- influence, a matrix with one row per observation and 4 columns, giving that observation's first derivative with respect to the count vector.

```
⟨concordance3⟩=
 #include "survS.h"
 #include "survproto.h"

 ⟨walkup⟩

 SEXP concordance3(SEXP y, SEXP x2, SEXP wt2, SEXP timewt2,
                       SEXP sortstop, SEXP doresid2) {
     int i, j, k, ii, jj, kk, j2;
     int n, ntree, nevent;
     double *time, *status;
     int xsave;

     /* sum of weights for a node (nwt), sum of weights for the node and
     **  all of its children (twt), then the same again for the subset of
     **  deaths
     */
     double *nwt, *twt, *dnwt, *dtwt;
     double z2;  /* sum of z^2 values */

     int ndeath;   /* total number of deaths at this point */
     int utime;    /* number of unique event times seen so far */
```

```
double dwt, dwt2;   /* sum of weights for deaths and deaths tied on x */
double wsum[3]; /* the sum of weights that are > current, <, or equal  */
double adjtimewt;  /* accounts for npair and timewt*/

SEXP rlist, count2, imat2, resid2;
double *count, *imat[5], *resid[3];
double *wt, *timewt;
int    *x, *sort2;
int doresid;
static const char *outnames1[]={"count", "influence", "resid", ""},
                  *outnames2[]={"count", "influence", ""};

n = nrows(y);
doresid = asLogical(doresid2);
x = INTEGER(x2);
wt = REAL(wt2);
timewt = REAL(timewt2);
sort2 = INTEGER(sortstop);
time = REAL(y);
status = time + n;

/* if there are tied predictors, the total size of the tree will be < n */
ntree =0; nevent =0;
for (i=0; i<n; i++) {
    if (x[i] >= ntree) ntree = x[i] +1;
    nevent += status[i];
}

nwt = (double *) R_alloc(4*ntree, sizeof(double));
twt = nwt + ntree;
dnwt = twt + ntree;
dtwt = dnwt + ntree;

for (i=0; i< 4*ntree; i++) nwt[i] =0.0;

if (doresid) PROTECT(rlist = mkNamed(VECSXP, outnames1));
else  PROTECT(rlist = mkNamed(VECSXP, outnames2));
count2 = SET_VECTOR_ELT(rlist, 0, allocVector(REALSXP, 6));
count = REAL(count2);
for (i=0; i<6; i++) count[i]=0.0;
imat2 = SET_VECTOR_ELT(rlist, 1, allocMatrix(REALSXP, n, 5));
for (i=0; i<5; i++) {
    imat[i] = REAL(imat2) + i*n;
    for (j=0; j<n; j++) imat[i][j] =0;
}
if (doresid==1) {
        resid2 = SET_VECTOR_ELT(rlist, 2, allocMatrix(REALSXP, nevent, 3));
        for (i=0; i<3; i++) resid[i] = REAL(resid2) + i*nevent;
        }

    ⟨concordance3-work⟩

    UNPROTECT(1);
    return(rlist);
}
```

The key part of our computation is to update the vectors of weights. We don't actually pass the risk score values r into the routine, it is enough for each observation to point to the appropriate tree node. The tree contains the weights for everyone whose survival is larger than the time currently under review, so starts with all weights equal to zero. For any pair of observations i, j we need to add $w_i w_j$ to the appropriate count, w_j to subject i's row of the leverage matrix and w_i to subject j's row. We use two trees to do this efficiently, one with all the observations to date, one with the events to date. Starting at the largest time (which is sorted last), walk through the tree.

- If the current observation is a censoring time, in order:
  - Subtract event tree information from the influence matrix
  - Update the Cox variance
  - Add them into the main tree
- If the current observation is a death, care for all deaths tied at this time point. Each pass covers all the deaths.
  - Pass 1: In any order
    - Add up the total number of deaths
    - Update the tied.y count and tied.xy count
      tied.xy subtotals reset each time x changes
    - Count concordant, discordant, tied.x counts, both total and for the observation's influence
    - Add the subject to the event tree
  - Finish up the tied.xy influence, for the last unique x in this set.
  - Pass 2:
    - Subtract the event tree information from the influence matrix
    - Add the tied.y part of the influence for each obs
    - Increment the Cox variance
    - Add the subject into the main tree
  - Pass 3: compute the residuals.
- When all the subjects have been added to the tree, then add the final death tree's data for to the influence matrix.

For concordant, discordant, and tied.x there are three readouts: the total tree before any additions, the death tree after the addition of the tied events, and the death tree at the very end. Increments to the Cox variance occur just before each addition to the total tree, and are saved out after each batch of events.

The above discussion counts up all pairs that are not tied on the response y. Though not used in the concordance the routine counts up tied.y pairs as well, with a separate count for those that are tied on both x and y. The algorithm for this part is simpler since the data is sorted by y. Say that there were 5 obs tied at some time point with weights of w_1 to w_5. The total count for ties involves all 5-choose-2 pairs and can be written as

$$w_1 w_2 + (w_1 + w_2)w_3 + (w_1 + w_2 + w_3)w_4 + (w_1 + w_2 + w_3 + w_4)w_5$$

which immediately suggests a simple summation algorithm as we go through the loop. In the below `dwt` contains the running sum 0, w_1, $w_1 + w_2$, etc and we add `w[i]*dwt` to the total just before incrementing the sum. The influence for observation 1 is $w_2 + w_3 + w_4 + w_5$, which can be done at the end as `dwt - wt[i]`. The temporary accumulator `dwt` is reset to 0 with each new y value. To compute ties on both x and y the data set is sorted by x within y, and we use the same algorithm, but reset `dwt2` to zero whenever either x or y changes.

```
⟨concordance3-work⟩=
 z2 =0; utime=0;
 for (i=0; i<n;) {
     ii = sort2[i];
     if (status[ii]==0) { /* censored, simply add them into the tree */
         /* Initialize the influence */
         walkup(dnwt, dtwt, x[ii], wsum, ntree);
         imat[0][ii] -= wsum[1];
         imat[1][ii] -= wsum[0];
         imat[2][ii] -= wsum[2];

         /* Cox variance */
         walkup(nwt, twt, x[ii], wsum, ntree);
         z2 += wt[ii]*(wsum[0]*(wt[ii] + 2*(wsum[1] + wsum[2])) +
                       wsum[1]*(wt[ii] + 2*(wsum[0] + wsum[2])) +
                       (wsum[0]-wsum[1])*(wsum[0]-wsum[1]));
         /* add them to the tree */
         addin(nwt, twt, x[ii], wt[ii]);
         i++;
     }
     else {  /* process all tied deaths at this point */
         ndeath=0; dwt=0;
         dwt2 =0; xsave=x[ii]; j2= i;
         adjtimewt = timewt[utime++];

         /* pass 1 */
         for (j=i; j<n && time[sort2[j]]==time[ii]; j++) {
             jj = sort2[j];
```

```
    ndeath++;
    count[3] += wt[jj] * dwt * adjtimewt;  /* update total tied on y */
    dwt += wt[jj];   /* sum of wts at this death time */

    if (x[jj] != xsave) {  /* restart the tied.xy counts */
        if (wt[sort2[j2]] < dwt2) { /* more than 1 tied */
            for (; j2<j; j2++) {
                /* update influence for this subgroup of x */
                kk = sort2[j2];
                imat[4][kk] += (dwt2- wt[kk]) * adjtimewt;
                imat[3][kk] -= (dwt2- wt[kk]) * adjtimewt;
            }
        } else j2 = j;
        dwt2 =0;
        xsave = x[jj];
    }
    count[4] += wt[jj] * dwt2 * adjtimewt; /* tied on xy */
    dwt2 += wt[jj]; /* sum of tied.xy weights */

    /* Count concordant, discordant, etc. */
    walkup(nwt, twt, x[jj], wsum, ntree);
    for (k=0; k<3; k++) {
        count[k] += wt[jj]* wsum[k] * adjtimewt;
        imat[k][jj] += wsum[k]*adjtimewt;
    }

    /* add to the event tree */
    addin(dnwt, dtwt, x[jj], adjtimewt*wt[jj]);  /* weighted deaths */

}
/* finish the tied.xy influence */
if (wt[sort2[j2]] < dwt2) { /* more than 1 tied */
    for (; j2<j; j2++) {
        /* update influence for this subgroup of x */
        kk = sort2[j2];
        imat[4][kk] += (dwt2- wt[kk]) * adjtimewt;
        imat[3][kk] -= (dwt2- wt[kk]) * adjtimewt;
    }
}

/* pass 2 */
for (j=i; j< (i+ndeath); j++) {
    jj = sort2[j];
    /* Update influence */
    walkup(dnwt, dtwt, x[jj], wsum, ntree);
    imat[0][jj] -= wsum[1];
```

```
            imat[1][jj] -= wsum[0];
            imat[2][jj] -= wsum[2];  /* tied.x */
            imat[3][jj] += (dwt- wt[jj])* adjtimewt;

            /* increment Cox var and add obs into the tree */
            walkup(nwt, twt, x[jj], wsum, ntree);
            z2 += wt[jj]*(wsum[0]*(wt[jj] + 2*(wsum[1] + wsum[2])) +
                          wsum[1]*(wt[jj] + 2*(wsum[0] + wsum[2])) +
                          (wsum[0]-wsum[1])*(wsum[0]-wsum[1]));

            addin(nwt, twt, x[jj], wt[jj]);
        }
        count[5] += dwt * adjtimewt* z2/twt[0]; /* weighted var in risk set*/

        /*
        ** Residuals are done after the deaths have been added to the tree
        **   since they are based on the Cox model risk set
        */
        if (doresid) {
            for (j=i; j< (i+ndeath); j++) {
                jj = sort2[j];
                walkup(nwt, twt, x[jj], wsum, ntree);
                nevent--;
                resid[0][nevent] = (wsum[0] - wsum[1])/twt[0]; /* -1 to 1 */
                resid[1][nevent] = twt[0] * adjtimewt;
                resid[2][nevent] = wt[jj];
            }
        }
        i += ndeath;
    }
}

/*
** Now finish off the influence for each observation
**  Since times flip (looking backwards) the wsum contributions flip too
*/
for (i=0; i<n; i++) {
    ii = sort2[i];
    walkup(dnwt, dtwt, x[ii], wsum, ntree);
    imat[0][ii] += wsum[1];
    imat[1][ii] += wsum[0];
    imat[2][ii] += wsum[2];
}
count[3] -= count[4];   /* the tied.xy were counted twice, once as tied.y */
```

⟨*walkup*⟩=

```
/*
** Given a tree described by
**   nwt = weight at each node
**   twt = weight of node + children
**   index = pointer to a location in the tree
**   ntree = number of nodes in the tree
** return the  count of those smaller, greater, tied
*/
void walkup(double *nwt, double* twt, int index, double sums[3], int ntree) {
    int i, j, parent;

    for (i=0; i<3; i++) sums[i] = 0.0;
    sums[2] = nwt[index];   /* tied on x */

    j = 2*index +2;  /* right child */
    if (j < ntree) sums[0] += twt[j];
    if (j <=ntree) sums[1]+= twt[j-1]; /*left child */

    while(index > 0) { /* for as long as I have a parent... */
        parent = (index-1)/2;
        if (index%2 == 1) sums[0] += twt[parent] - twt[index]; /* left child */
        else sums[1] += twt[parent] - twt[index]; /* I am a right child */
        index = parent;
    }
}

/* Add an observation into the tree (a negative weight takes them out) */
void addin(double *nwt, double *twt, int index, double wt) {
    nwt[index] += wt;
    while (index >0) {
        twt[index] += wt;
        index = (index-1)/2;
    }
    twt[0] += wt;
}
```

The code for [start, stop) data is almost identical, the primary call simply has one more index. As in the agreg routines there are two sort indices, the first indexes the data by stop time, longest to earliest, and the second by start time. The y variable now has three columns.

⟨*concordance3*⟩=

```
SEXP concordance4(SEXP y, SEXP x2, SEXP wt2, SEXP timewt2,
                  SEXP sortstart, SEXP sortstop, SEXP doresid2) {
int i, j, k, ii, jj, kk, i2, j2;
int n, ntree, nevent;
double *time1, *time2, *status;
int xsave;
```

```
/* sum of weights for a node (nwt), sum of weights for the node and
**  all of its children (twt), then the same again for the subset of
**  deaths
*/
double *nwt, *twt, *dnwt, *dtwt;
double z2;  /* sum of z^2 values */

int ndeath;   /* total number of deaths at this point */
int utime;    /* number of unique event times seen so far */
double dwt;   /* weighted number of deaths at this point */
double dwt2;  /* tied on both x and y */
double wsum[3]; /* the sum of weights that are > current, <, or equal  */
double adjtimewt;  /* accounts for npair and timewt*/

SEXP rlist, count2, imat2, resid2;
double *count, *imat[5], *resid[3];
double *wt, *timewt;
int    *x, *sort2, *sort1;
int doresid;
static const char *outnames1[]={"count", "influence", "resid", ""},
                  *outnames2[]={"count", "influence", ""};

n = nrows(y);
doresid = asLogical(doresid2);
x = INTEGER(x2);
wt = REAL(wt2);
timewt = REAL(timewt2);
sort2 = INTEGER(sortstop);
sort1 = INTEGER(sortstart);
time1 = REAL(y);
time2 = time1 + n;
status = time2 + n;

/* if there are tied predictors, the total size of the tree will be < n */
ntree =0; nevent =0;
for (i=0; i<n; i++) {
    if (x[i] >= ntree) ntree = x[i] +1;
    nevent += status[i];
}

/*
** nwt and twt are the node weight and total =node + all children for the
**  tree holding all subjects.  dnwt and dtwt are the same for the tree
**  holding all the events
*/
```

```
    nwt = (double *) R_alloc(4*ntree, sizeof(double));
    twt = nwt + ntree;
    dnwt = twt + ntree;
    dtwt = dnwt + ntree;

    for (i=0; i< 4*ntree; i++) nwt[i] =0.0;

    if (doresid) PROTECT(rlist = mkNamed(VECSXP, outnames1));
    else  PROTECT(rlist = mkNamed(VECSXP, outnames2));
    count2 = SET_VECTOR_ELT(rlist, 0, allocVector(REALSXP, 6));
    count = REAL(count2);
    for (i=0; i<6; i++) count[i]=0.0;
    imat2 = SET_VECTOR_ELT(rlist, 1, allocMatrix(REALSXP, n, 5));
    for (i=0; i<5; i++) {
        imat[i] = REAL(imat2) + i*n;
        for (j=0; j<n; j++) imat[i][j] =0;
    }
    if (doresid==1) {
        resid2 = SET_VECTOR_ELT(rlist, 2, allocMatrix(REALSXP, nevent, 3));
        for (i=0; i<3; i++) resid[i] = REAL(resid2) + i*nevent;
        }

    ⟨concordance4-work⟩

    UNPROTECT(1);
    return(rlist);
}
```

As we move from the longest time to the shortest observations are added into the tree of weights whenever we encounter their stop time. This is just as before. Weights now also need to be removed from the tree whenever we encounter an observation's start time. It is convenient "catch up" on this second task whenever we encounter a death.

```
⟨concordance4-work⟩=
 z2 =0; utime=0; i2 =0;  /* i2 tracks the start times */
 for (i=0; i<n;) {
     ii = sort2[i];
     if (status[ii]==0) { /* censored, simply add them into the tree */
         /* Initialize the influence */
         walkup(dnwt, dtwt, x[ii], wsum, ntree);
         imat[0][ii] -= wsum[1];
         imat[1][ii] -= wsum[0];
         imat[2][ii] -= wsum[2];

         /* Cox variance */
         walkup(nwt, twt, x[ii], wsum, ntree);
         z2 += wt[ii]*(wsum[0]*(wt[ii] + 2*(wsum[1] + wsum[2])) +
                       wsum[1]*(wt[ii] + 2*(wsum[0] + wsum[2])) +
                       (wsum[0]-wsum[1])*(wsum[0]-wsum[1]));
         /* add them to the tree */
         addin(nwt, twt, x[ii], wt[ii]);
         i++;
     }
     else {  /* a death */
         /* remove any subjects whose start time has been passed */
         for (; i2<n && (time1[sort1[i2]] >= time2[ii]); i2++) {
             jj = sort1[i2];
             /* influence */
             walkup(dnwt, dtwt, x[jj], wsum, ntree);
             imat[0][jj] += wsum[1];
             imat[1][jj] += wsum[0];
             imat[2][jj] += wsum[2];

             addin(nwt, twt, x[jj], -wt[jj]);  /*remove from main tree */

             /* Cox variance */
             walkup(nwt, twt, x[jj], wsum, ntree);
             z2 -= wt[jj]*(wsum[0]*(wt[jj] + 2*(wsum[1] + wsum[2])) +
                           wsum[1]*(wt[jj] + 2*(wsum[0] + wsum[2])) +
                           (wsum[0]-wsum[1])*(wsum[0]-wsum[1]));
         }

         ndeath=0; dwt=0;
         dwt2 =0; xsave=x[ii]; j2= i;
         adjtimewt = timewt[utime++];

         /* pass 1 */
         for (j=i; j<n && (time2[sort2[j]]==time2[ii]); j++) {
             jj = sort2[j];
             ndeath++;
             jj = sort2[j];
             count[3] += wt[jj] * dwt;  /* update total tied on y */
             dwt += wt[jj];   /* count of deaths and sum of wts */

             if (x[jj] != xsave) {  /* restart the tied.xy counts */
                 if (wt[sort2[j2]] < dwt2) { /* more than 1 tied */
                     for (; j2<j; j2++) {
                         /* update influence for this subgroup of x */
                         kk = sort2[j2];
                         imat[4][kk] += (dwt2- wt[kk]) * adjtimewt;
                         imat[3][kk] -= (dwt2- wt[kk]) * adjtimewt;
                     }
                 } else j2 = j;
                 dwt2 =0;
                 xsave = x[jj];
             }
             count[4] += wt[jj] * dwt2 * adjtimewt; /* tied on xy */
             dwt2 += wt[jj]; /* sum of tied.xy weights */

             /* Count concordant, discordant, etc. */
             walkup(nwt, twt, x[jj], wsum, ntree);
             for (k=0; k<3; k++) {
                 count[k] += wt[jj]* wsum[k] * adjtimewt;
                 imat[k][jj] += wsum[k]*adjtimewt;
             }

             /* add to the event tree */
             addin(dnwt, dtwt, x[jj], adjtimewt*wt[jj]);  /* weighted deaths */
         }
         /* finish the tied.xy influence */
         if (wt[sort2[j2]] < dwt2) { /* more than 1 tied */
             for (; j2<j; j2++) {
                 /* update influence for this subgroup of x */
                 kk = sort2[j2];
                 imat[4][kk] += (dwt2- wt[kk]) * adjtimewt;
                 imat[3][kk] -= (dwt2- wt[kk]) * adjtimewt;
             }
         }

         /* pass 3 */
         for (j=i; j< (i+ndeath); j++) {
             jj = sort2[j];
             /* Update influence */
             walkup(dnwt, dtwt, x[jj], wsum, ntree);
             imat[0][jj] -= wsum[1];
             imat[1][jj] -= wsum[0];
             imat[2][jj] -= wsum[2];  /* tied.x */
             imat[3][jj] += (dwt- wt[jj])* adjtimewt;

             /* increment Cox var and add obs into the tree */
             walkup(nwt, twt, x[jj], wsum, ntree);
             z2 += wt[jj]*(wsum[0]*(wt[jj] + 2*(wsum[1] + wsum[2])) +
                           wsum[1]*(wt[jj] + 2*(wsum[0] + wsum[2])) +
                           (wsum[0]-wsum[1])*(wsum[0]-wsum[1]));

             addin(nwt, twt, x[jj], wt[jj]);
         }
         count[5] += dwt * adjtimewt* z2/twt[0]; /* weighted var in risk set*/
```

```
        if (doresid) {
            for (j=i; j< (i+ndeath); j++) {
                jj = sort2[j];
                walkup(nwt, twt, x[jj], wsum, ntree);
                nevent--;
                resid[0][nevent] = (wsum[0] - wsum[1])/twt[0]; /* -1 to 1 */
                resid[1][nevent] = twt[0] * adjtimewt;
                resid[2][nevent] = wt[jj];
            }
        }
        i += ndeath;

    }
}

/*
** Now finish off the influence for those not yet removed
**  Since times flip (looking backwards) the wsum contributions flip too
*/
for (; i2<n; i2++) {
    ii = sort1[i2];
    walkup(dnwt, dtwt, x[ii], wsum, ntree);
    imat[0][ii] += wsum[1];
    imat[1][ii] += wsum[0];
    imat[2][ii] += wsum[2];
}
count[3] -= count[4]; /* tied.y was double counted a tied.xy */
```

5 Expected Survival

The expected survival routine creates the overall survival curve for a *group* of people. It is possible to take the set of expected survival curves for each individual and average them, which is the `Ederer` method below, but this is not always the wisest choice: the Hakulinen and conditional methods average in anothers ways, both of which are more sophisticated ways to deal with censoring. The individual curves are dervived either from population rate tables such as the US annual life tables from the National Center for Health Statistics or the larger multi-national collection at mortality.org, or by using a previously fitted Cox model as the table.

The arguments for `survexp` are

formula The model formula. The right hand side consists of grouping variables, identically to [[survfit]] and an optional [[ratetable]] directive. The "response" varies by method:

- for the Hakulinen method it is a vector of censoring times. This is the actual censoring time for censored subjecs, and is what the censoring time 'would have been' for each subject who died.
- for the conditional method it is the usual Surv(time, status) code

- for the Ederer method no response is needed

data, weights, subset, na.action as usual

rmap an optional mapping for rate table variables, see more below.

times An optional vector of time points at which to compute the response. For the Hakulinen and conditional methods the program uses the vector of unique y values if this is missing. For the Ederer the component is not optional.

method The method used for the calculation. Choices are individual survival, or the Ederer, Hakulinen, or conditional methods for cohort survival.

cohort, conditional Older arguments that were used to select the method.

ratetable the population rate table to use as a reference. This can either be a ratetable object or a previously fitted Cox model

scale Scale the resulting output times, e.g., 365.25 to turn days into years.

se.fit This has been deprecated.

model, x, y usual

The output of survexp contains a subset of the elements in a `survfit` object, so many of the survfit methods can be applied. The result has a class of `c('survexp', 'survfit')`.

```
⟨survexp⟩=
 survexp <- function(formula, data,
         weights, subset, na.action, rmap, times,
         method=c("ederer", "hakulinen", "conditional", "individual.h",
                  "individual.s"),
         cohort=TRUE,  conditional=FALSE,
         ratetable=survival::survexp.us, scale=1, se.fit,
         model=FALSE, x=FALSE, y=FALSE) {
     ⟨survexp-setup⟩
     ⟨survexp-compute⟩
     ⟨survexp-format⟩
     ⟨survexp-finish⟩
 }
```

The first few lines are standard. Keep a copy of the call, then manufacture a call to `model.frame` that contains only the arguments relevant to that function.

```
⟨survexp-setup⟩=
 Call <- match.call()

 # keep the first element (the call), and the following selected arguments
 indx <- match(c('formula', 'data', 'weights', 'subset', 'na.action'),
                   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

Terms <- if(missing(data)) terms(formula, 'ratetable')
         else              terms(formula, 'ratetable',data=data)
```

The function works with two data sets, the user's data on an actual set of subjects and the reference ratetable. This leads to a particular nuisance, that the variable names in the data set may not match those in the ratetable. For instance the United States overall death rate table `survexp.us` expects 3 variables, as shown by `summary(survexp.us)`

- age = age in days for each subject at the start of follow-up
- sex = sex of the subject, "male" or "female" (the routine accepts any unique abbreviation and is case insensitive)
- year = date of the start of follow-up

Up until the most recent revision, the formula contained any necessary mapping between the variables in the data set and the ratetable. For instance

```
 survexp( ~ sex + ratetable(age=age*365.25, sex=sex,
                           year=entry.dt),
            data=mydata, ratetable=survexp.us)
```

In this case the user's data set has a variable 'age' containing age in years, along with sex and an entry date. This had to be changed for two reasons. The primary one is that the data in a `ratetable` call had to be converted into a matrix in order to "pass through" the model.frame logic. With the recent updates to coxph so that it remembers factor codings correctly in new data sets, it is advantageous to keep factors as factors. The second is that a coxph model with a large number of covariates induces a very long ratetable clause; at about 40 variable it caused one of the R internal routines to fail due to a long expression. A third reason, perhaps the most pressing in reality, is that I've always felt that the prior code was confusing since it used the same term 'ratetable' for two different tasks.

The new process adds the `rmap` argument, an example would be `rmap=list(age =age*365.25, year=entry.dt)`. Any variables in the ratetable that are not found in `rmap` are assumed to not need a mapping, this would be `sex` in the above example. For backwards compatability we allow the old style argument, converting it into the new style.

The `rmap` argument needs to be examined without evaluating it; we then add the appropriate extra variables into a temporary formula so that the model frame has all that is required. The ratetable variables then can be retrieved from the model frame. The `pyears` routine uses the same rmap argument; this segment of the code is given its own name so that it can be included there as well.

⟨*survexp-setup*⟩=
```
rate <- attr(Terms, "specials")$ratetable
if(length(rate) > 1)
        stop("Can have only 1 ratetable() call in a formula")
```

```
 ⟨survexp-setup-rmap⟩

 mf <- eval(tform, parent.frame())

⟨survexp-setup-rmap⟩=
 if(length(rate) == 1) {
     if (!missing(rmap))
         stop("The ratetable() call in a formula is depreciated")

     stemp <- untangle.specials(Terms, 'ratetable')
     rcall <- as.call(parse(text=stemp$var)[[1]])   # as a call object
     rcall[[1]] <- as.name('list')                  # make it a call to list(..
     Terms <- Terms[-stemp$terms]                   # remove from the formula
     }
 else if (!missing(rmap)) {
     rcall <- substitute(rmap)
     if (!is.call(rcall) || rcall[[1]] != as.name('list'))
         stop ("Invalid rcall argument")
     }
 else rcall <- NULL   # A ratetable, but no rcall argument

 # Check that there are no illegal names in rcall, then expand it
 #  to include all the names in the ratetable
 if (is.ratetable(ratetable))   {
     varlist <- names(dimnames(ratetable))
     if (is.null(varlist)) varlist <- attr(ratetable, "dimid") # older style
 }
 else if(inherits(ratetable, "coxph") && !inherits(ratetable, "coxphms")) {
     ## Remove "log" and such things, to get just the list of
     #   variable names
     varlist <- all.vars(delete.response(ratetable$terms))
     }
 else stop("Invalid rate table")

 temp <- match(names(rcall)[-1], varlist) # 2,3,... are the argument names
 if (any(is.na(temp)))
     stop("Variable not found in the ratetable:", (names(rcall))[is.na(temp)])

 if (any(!(varlist %in% names(rcall)))) {
     to.add <- varlist[!(varlist %in% names(rcall))]
     temp1 <- paste(text=paste(to.add, to.add, sep='='), collapse=',')
     if (is.null(rcall)) rcall <- parse(text=paste("list(", temp1, ")"))[[1]]
     else {
         temp2 <- deparse(rcall)
         rcall <- parse(text=paste("c(", temp2, ",list(", temp1, "))"))[[1]]
         }
```

```
    }
```

The formula below is used only in the call to `model.frame` to ensure that the frame has both the formula and the ratetable variables. We don't want to modify the original formula, since we use it to create the X matrix and the response variable. The non-obvious bit of code is the addition of an environment to the formula. The `model.matrix` routine has a non-standard evaluation - it uses the frame of the formula, rather than the parent.frame() argument below, along with the `data` to look up variables. If a formula is long enough deparse() will give two lines, hence the extra paste call to re-collapse it into one.

⟨*survexp-setup-rmap*⟩=
```
# Create a temporary formula, used only in the call to model.frame
newvar <- all.vars(rcall)
if (length(newvar) > 0) {
    temp <- paste(paste(deparse(Terms), collapse=""),
                  paste(newvar, collapse='+'), sep='+')
    tform$formula <- as.formula(temp, environment(Terms))
    }
```

If the user data has 0 rows, e.g. from a mistaken `subset` statement that eliminated all subjects, we need to stop early. Otherwise the .C code fails in a nasty way.

⟨*survexp-setup*⟩=
```
n <- nrow(mf)
if (n==0) stop("Data set has 0 rows")
if (!missing(se.fit) && se.fit)
    warning("se.fit value ignored")

weights <- model.extract(mf, 'weights')
if (length(weights) ==0) weights <- rep(1.0, n)
if (inherits(ratetable, 'ratetable') && any(weights !=1))
    warning("weights ignored")

if (any(attr(Terms, 'order') >1))
        stop("Survexp cannot have interaction terms")
if (!missing(times)) {
    if (any(times<0)) stop("Invalid time point requested")
    if (length(times) >1 )
        if (any(diff(times)<0)) stop("Times must be in increasing order")
    }
```

If a response variable was given, we only need the times and not the status. To be correct, computations need to be done for each of the times given in the `times` argument as well as for each of the unique y values. This ends up as the vector `newtime`. If a `times` argument was given we will subset down to only those values at the end. For a population rate table and the Ederer method the times argument is required.

⟨*survexp-setup*⟩=
```
Y <- model.extract(mf, 'response')
```

```
no.Y <- is.null(Y)
if (no.Y) {
    if (missing(times)) {
        if (is.ratetable(ratetable))
            stop("either a times argument or a response is needed")
        }
    else newtime <- times
    }
else {
    if (is.matrix(Y)) {
        if (is.Surv(Y) && attr(Y, 'type')=='right') Y <- Y[,1]
        else stop("Illegal response value")
        }
    if (any(Y<0)) stop ("Negative follow up time")
#    if (missing(npoints)) temp <- unique(Y)
#    else                  temp <- seq(min(Y), max(Y), length=npoints)
    temp <- unique(Y)
    if (missing(times)) newtime <- sort(temp)
    else  newtime <- sort(unique(c(times, temp[temp<max(times)])))
    }

if (!missing(method)) method <- match.arg(method)
else {
    # the historical defaults and older arguments
    if (!missing(conditional) && conditional) method <- "conditional"
    else {
        if (no.Y) method <- "ederer"
        else method <- "hakulinen"
        }
    if (!missing(cohort) && !cohort) method <- "individual.s"
    }
if (no.Y && (method!="ederer"))
    stop("a response is required in the formula unless method='ederer'")
```

The next step is to check out the ratetable. For a population rate table a set of consistency checks is done by the `match.ratetable` function, giving a set of sanitized indices `R`. This function wants characters turned to factors. For a Cox model `R` will be a model matix whose covariates are coded in exactly the same way that variables were coded in the original Cox model. We call the model.matrix.coxph function to avoid repeating the steps found there (remove cluster statements, etc). We also need to use the `mf` argument of the function, otherwise it will call model.frame internally and fail when it can't find the response variable (which we don't need).

Note that for a population rate table the standard error of the expected is by definition 0 (the population rate table is based on a huge sample). For a Cox model rate table, an se formula is currently only available for the Ederer method.

⟨*survexp-compute*⟩=
```
ovars <- attr(Terms, 'term.labels')
```

```
# rdata contains the variables matching the ratetable
rdata <- data.frame(eval(rcall, mf), stringsAsFactors=TRUE)
if (is.ratetable(ratetable)) {
    israte <- TRUE
    if (no.Y) {
        Y <- rep(max(times), n)
        }
    rtemp <- match.ratetable(rdata, ratetable)
    R <- rtemp$R
    }
else if (inherits(ratetable, 'coxph')) {
    israte <- FALSE
    Terms <- ratetable$terms
#    if (!is.null(attr(Terms, 'offset')))
#        stop("Cannot deal with models that contain an offset")
#    strats <- attr(Terms, "specials")$strata
#    if (length(strats))
#        stop("survexp cannot handle stratified Cox models")
#
    if (any(names(mf[,rate]) !=  attr(ratetable$terms, 'term.labels')))
         stop("Unable to match new data to old formula")
    }
else if (inherits(ratetable, "coxphms"))
    stop("survexp not defined for multi-state coxph models")
else stop("Invalid ratetable")
```

Now for some calculation. If cohort is false, then any covariates on the right hand side (other than the rate table) are irrelevant, the function returns a vector of expected values rather than survival curves.

⟨*survexp-compute*⟩=

```
if (substring(method, 1, 10) == "individual") { #individual survival
    if (no.Y) stop("for individual survival an observation time must be given")
    if (israte)
         temp <- survexp.fit (1:n, R, Y, max(Y), TRUE, ratetable)
    else {
        rmatch <- match(names(data), names(rdata))
        if (any(is.na(rmatch))) rdata <- cbind(rdata, data[,is.na(rmatch)])
        temp <- survexp.cfit(1:n, rdata, Y, 'individual', ratetable)
    }
    if (method == "individual.s") xx <- temp$surv
    else xx <- -log(temp$surv)
    names(xx) <- row.names(mf)
    na.action <- attr(mf, "na.action")
    if (length(na.action)) return(naresid(na.action, xx))
    else return(xx)
    }
```

Now for the more commonly used case: returning a survival curve. First see if there are any grouping variables. The results of the `tcut` function are often used in person-years analysis, which is somewhat related to expected survival. However tcut results aren't relevant here and we put in a check for the confused user. The strata command creates a single factor incorporating all the variables.

⟨*survexp-compute*⟩=
```
if (length(ovars)==0)  X <- rep(1,n)  #no categories
else {
    odim <- length(ovars)
    for (i in 1:odim) {
        temp <- mf[[ovars[i]]]
        ctemp <- class(temp)
        if (!is.null(ctemp) && ctemp=='tcut')
            stop("Can't use tcut variables in expected survival")
        }
    X <- strata(mf[ovars])
    }

#do the work
if (israte)
    temp <- survexp.fit(as.numeric(X), R, Y, newtime,
                        method=="conditional", ratetable)
else {
    temp <- survexp.cfit(as.numeric(X), rdata, Y, method, ratetable, weights)
    newtime <- temp$time
    }
```

Now we need to package up the curves properly All the results can be returned as a single matrix of survivals with a common vector of times. If there was a times argument we need to subset to selected rows of the computation.

⟨*survexp-format*⟩=
```
if (missing(times)) {
    n.risk <- temp$n
    surv <- temp$surv
    }
else {
    if (israte) keep <- match(times, newtime)
    else {
        # The result is from a Cox model, and it's list of
        #  times won't match the list requested in the user's call
        # Interpolate the step function, giving survival of 1
        #  for requested points that precede the Cox fit's
        #  first downward step.  The code is like summary.survfit.
        n <- length(temp$time)
        keep <- approx(temp$time, 1:n, xout=times, yleft=0,
                       method='constant', f=0, rule=2)$y
        }

    if (is.matrix(temp$surv)) {
        surv <- (rbind(1,temp$surv))[keep+1,,drop=FALSE]
        n.risk <- temp$n[pmax(1,keep),,drop=FALSE]
         }
    else {
        surv <- (c(1,temp$surv))[keep+1]
        n.risk <- temp$n[pmax(1,keep)]
        }
    newtime <- times
    }
newtime <- newtime/scale
if (is.matrix(surv)) {
    dimnames(surv) <- list(NULL, levels(X))
    out <- list(call=Call, surv= drop(surv), n.risk=drop(n.risk),
                    time=newtime)
    }
else {
     out <- list(call=Call, surv=c(surv), n.risk=c(n.risk),
                   time=newtime)
     }
```

Last do the standard things: add the model, x, or y components to the output if the user asked for them. (For this particular routine I can't think of a reason they every would.) Copy across summary information from the rate table computation if present, and add the method and class to the output.

```
⟨survexp-finish⟩=
 if (model) out$model <- mf
 else {
     if (x) out$x <- X
     if (y) out$y <- Y
     }
 if (israte && !is.null(rtemp$summ)) out$summ <- rtemp$summ
 if (no.Y) out$method <- 'Ederer'
 else if (conditional) out$method <- 'conditional'
 else                  out$method <- 'cohort'
 class(out) <- c('survexp', 'survfit')
 out
```

5.1 Parsing the covariates list

For a multi-state Cox model we allow a list of formulas to take the place of the `formula` argument. The first element of the list is the default formula, later elements are of the form `transitions` `formula/options`, where the left hand side denotes one or more transitions, and the right hand side is used to augment the basic formula wrt those transitions.

Step 1 is to break the formula into parts. There will be a list of left sides, a list of right sides, and a list of options. From this we can create a single “pseudo formula” that is used to drive the model.frame process, which ensures that all of the variables we need will be found in the model frame. Further processing has to wait until after the model frame has been constructed, i.e., if a left side referred to state “deathh” that might be a real state or a typing mistake, we can’t know until the data is in hand.

Should we walk the parse tree of the formula, or convert it to character and use string manipulations? The latter looks promising until you see a fragment like this: `entry:death age/sex + ns(weight/height, df=4) / common` Walking the parse tree is a bit more subtle, but we then can take advantage of all the knowledge built into the R parser. A formula is a 3 element list of “ ”, leftside, rightside, or 2 elements if it has only a right hand side. Legal ones for coxph have both left and right.

```
⟨parsecovar⟩=
 parsecovar1 <- function(flist, statedata) {
     if (any(sapply(flist, function(x) !inherits(x, "formula"))))
         stop("an element of the formula list is not a formula")
     if (any(sapply(flist, length) != 3))
         stop("all formulas must have a left and right side")

     # split the formulas into a right hand and left hand side
     lhs <- lapply(flist, function(x) x[-3])   # keep the ~
     rhs <- lapply(flist, function(x) x[[3]])  # don’t keep the ~

     rhs <- parse_rightside(rhs)
     ⟨parse-leftside⟩
     list(rhs = rhs, lhs= lterm)
 }
```

Figure 2 shows the parse tree for a complex formula. The following function splits the formula at the rightmost slash, ignoring the inside of any function or parenthesised phrase. Recursive functions like this are almost impossible to read, but luckily it is short. The formula recurrs on the left and right side of +*: and %in%, and on binary - (but not on unary -).

```
⟨parsecovar⟩=
 rightslash <- function(x) {
     if (!inherits(x, ’call’)) return(x)
     else {
         if (x[[1]] == as.name(’/’)) return(list(x[[2]], x[[3]]))
         else if (x[[1]]==as.name(’+’) || (x[[1]]==as.name(’-’) && length(x)==3)||
                  x[[1]]==as.name(’*’) || x[[1]]==as.name(’:’)  ||
                  x[[1]]==as.name(’%in%’)) {
                      temp <- rightslash(x[[3]])
                      if (is.list(temp)) {
                          x[[3]] <- temp[[1]]
```

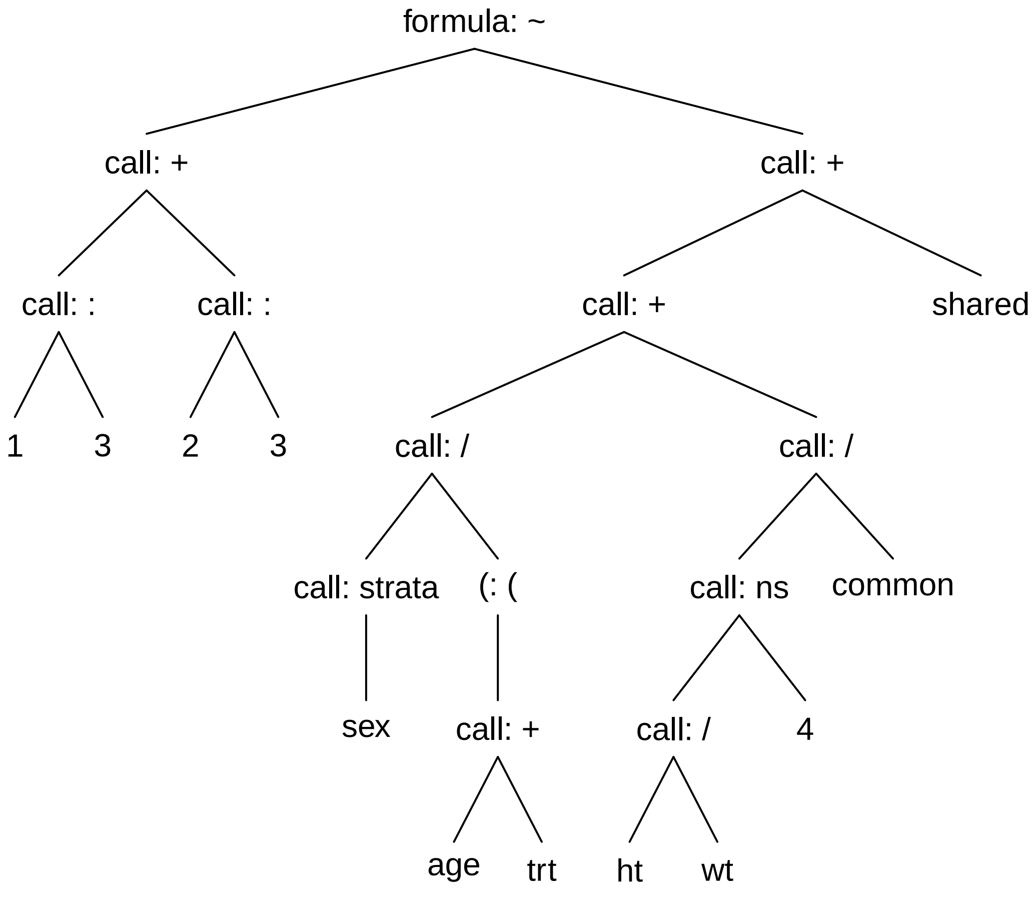

Figure 2: The parse tree for the formula `1:3 +2:3  strata(sex)/(age + trt) + ns(weight/ht, df=4) / common + shared`

```
                          return(list(x, temp[[2]]))
                      } else {
                          temp <- rightslash(x[[2]])
                          if (is.list(temp)) {
                              x[[2]] <- temp[[2]]
                              return(list(temp[[1]], x))
                          } else return(x)
                      }
                  }
         else return(x)
     }
 }
```

There are 4 possble options of common, shared, and init. The first 2 appear just as words, the last should have a set of values attached which become the `ival` vector. There will, of course, one day be a user with a variable named `common` who wants a nested term `x/common`. Since we don't look inside parenthesis they will be able to use `1:3    (x/common)`.

```
⟨parsecovar⟩=
 parse_rightside <- function(rhs) {
     parts <- lapply(rhs, rightslash)
     new <- lapply(parts, function(opt) {
         tform <- ~ x    # a skeleton, "x" will be replaced
         if (!is.list(opt)) { # no options for this line
             tform[[2]] <- opt
             list(formula = tform, ival = NULL, common = FALSE,
                  shared = FALSE)
         }
         else{
             # treat the option list as though it were a formula
             temp <- ~ x
             temp[[2]] <- opt[[2]]
             optterms <- terms(temp)
             ff <- rownames(attr(optterms, "factors"))
             index <- match(ff, c("common", "shared", "init"))
             if (any(is.na(index)))
                 stop("option not recognized in a covariates formula: ",
                      paste(ff[is.na(index)], collapse=", "))
             common <- any(index==1)
             shared  <- any(index==2)
             if (any(index==3)) {
                 optatt <- attributes(optterms)
                 j <- optatt$variables[1 + which(index==3)]
                 j[[1]] <- as.name("list")
                 ival <- unlist(eval(j, parent.frame()))
             }
             else ival <- NULL
```

```
            tform[[2]] <- opt[[1]]
            list(formula= tform, ival= ival, common= common, shared=shared)
        }
    })
    new
}
```

The left hand side of each formula specifies the set of transitions to which the covariates apply, and is more complex. Say instance that we had 7 states and the following statedata data set.

state	A	N	death
A-N-	0	0	0
A+N-	1	0	0
A-N1	0	1	0
A+N1	1	1	0
A-N2	0	2	0
A+N2	1	2	0
Death	NA	NA	1

Here are some valid transitions

1. 0:state(’A+N+’), any transition to the A+N+ state
2. state(’A-N-’):death(0), a transition from A-N-, but not to death
3. A(0):A(1), any of the 4 changes that start with A=0 and end with A=1
4. N(0):N(1,2) + N(1):N(2), an upward change of N
5. ’A-N-’:c(’A-N+’,’A+N-’); if there is no variable then the overall state is assumed
6. 1:3 + 2:3; we can refer to states by number, and we can have multiples

⟨*parse-leftside*⟩=

```
# deal with the left hand side of the formula
# the next routine cuts at ’+’ signs
pcut <- function(form) {
    if (length(form)==3) {
        if (form[[1]] == ’+’)
            c(pcut(form[[2]]), pcut(form[[3]]))
        else if (form[[1]] == ’~’) pcut(form[[2]])
        else list(form)
    }
    else list(form)
}
lcut <- lapply(lhs, function(x) pcut(x[[2]]))
```

We now have one list per formula, each list is either a single term or a list of terms (case 4 above). To make evaluation easier, create functions that append their name to a list of values. I have not yet found a way to do this without eval(parse()), which always seems clumsy. A use for the labels without an argument will arise later, hence the double environments.

Repeating the list above, this is what we want to end with

- a list with one element per formula in the covariates list
- each element is a list, with one element per term: multiple a:b terms are allowed separated by + signs
- each of these level 3 elements is a list with two elements "left" and "right", for the two sides of the : operator
- left and right will be one of 3 forms: a simple vector, a one element list containing the stateid, or a two element list containing the stateid and the values. Any word that doesn't match one of the column names of statedata ends up as a vector.

```
⟨parse-leftside⟩=
 env1 <- new.env(parent= parent.frame(2))
 env2 <- new.env(parent= env1)
 if (missing(statedata)) {
     assign("state", function(...) list(stateid= "state",
                                        values=c(...)), env1)
     assign("state", list(stateid="state"))
 }
 else {
     for (i in statedata) {
         assign(i, eval(list(stateid=i)), env2)
         tfun <- eval(parse(text=paste0("function(...) list(stateid='"
                                        , i, "', values=c(...))")))
         assign(i, tfun, env1)
     }
 }
 lterm <- lapply(lcut, function(x) {
     lapply(x, function(z) {
         if (length(z)==1) {
             temp <- eval(z, envir= env2)
             if (is.list(temp) && names(temp)[[1]] =="stateid") temp
             else temp
         }
         else if (length(z) ==3 && z[[1]]==':')
             list(left=eval(z[[2]], envir=env2), right=eval(z[[3]], envir=env2))
         else stop("invalid term: ", deparse(z))
     })
 })
```

The second call, which builds tmap, the terms map. Arguments are the results from the first pass, the statedata data frame, the default formula, the terms structure from the full formula, and the transitions count.

One nuisance is that the terms function sometimes inverts things. For example in the formula `terms(  x1 + x1:iage + x2 + x2:iage)` the label for the second of these becomes `iage:x2`. I'm guessing it is because the variable first appear in the order x1, iage, x2 and labels make use of that order. But when we look at the formula fragment `x2 + x2:iage` the terms will be in the other order. A way out of this is to use the simple `termmatch` function below, which keys off of the factors attribute instead of the names.

```
⟨parsecovar⟩=
termmatch <- function(f1, f2) {
    # look for f1 in f2, each the factors attribute of a terms object
    if (length(f1)==0) return(NULL)   # a formula with only ~1
    irow <- match(rownames(f1), rownames(f2))
    if (any(is.na(irow))) stop ("termmatch failure 1")
    hashfun <- function(j) sum(ifelse(j==0, 0, 2^(seq(along.with=j))))
    hash1 <- apply(f1, 2, hashfun)
    hash2 <- apply(f2[irow,,drop=FALSE], 2, hashfun)
    index <- match(hash1, hash2)
    if (any(is.na(index))) stop("termmatch failure 2")
    index
}

parsecovar2 <- function(covar1, statedata, dformula, Terms, transitions,states) {
    if (is.null(statedata))
        statedata <- data.frame(state = states, stringsAsFactors=FALSE)
    else {
        if (is.null(statedata$state))
            stop("the statedata data set must contain a variable 'state'")
        indx1 <- match(states, statedata$state, nomatch=0)
        if (any(indx1==0))
            stop("statedata does not contain all the possible states: ",
                 states[indx1==0])
        statedata <- statedata[indx1,]   # put it in order
    }

    # Statedata might have rows for states that are not in the data set,
    #  for instance if the coxph call had used a subset argument.  Any of
    #  those were eliminated above.
    # Likewise, the formula list might have rules for transitions that are
    #  not present.  Don't worry about it at this stage.
    allterm <- attr(Terms, 'factors')
    nterm <- ncol(allterm)

    # create a map for every transition, even ones that are not used.
```

```
     # at the end we will thin it out
     # It has an extra first row for intercept (baseline)
     # Fill it in with the default formula
     nstate <- length(states)
     tmap <- array(0L, dim=c(nterm+1, nstate, nstate))
     dmap <- array(seq_len(length(tmap)), dim=c(nterm+1, nstate, nstate)) #unique values
     dterm <- termmatch(attr(terms(dformula), "factors"), allterm)
     dterm <- c(1L, 1L+ dterm)  # add intercept
     tmap[dterm,,] <- dmap[dterm,,]
     inits <- NULL

     if (!is.null(covar1)) {
         ⟨parse-tmap⟩
     }
     ⟨parse-finish⟩
 }
```

Now go through the formulas one by one. The left hand side tells us which state:state transitions to fill in, the right hand side tells the variables. The code block below goes through lhs element(s) for a single formula. That element is itself a list which has an entry for each term, and that entry can have left and right portions.

```
⟨parse-lmatch⟩=
 state1 <- state2 <- NULL
 for (x in lhs) {
     # x is one term
     if (!is.list(x) || is.null(x$left)) stop("term found without a ':' ", x)
     # left of the colon
     if (!is.list(x$left) && length(x$left) ==1 && x$left==0)
         temp1 <- 1:nrow(statedata)
     else if (is.numeric(x$left)) {
         temp1 <- as.integer(x$left)
         if (any(temp1 != x$left)) stop("non-integer state number")
         if (any(temp1 <1 | temp1> nstate))
             stop("numeric state is out of range")
     }
     else if (is.list(x$left) && names(x$left)[1] == "stateid"){
         if (is.null(x$left$value))
             stop("state variable with no list of values: ",x$left$stateid)
         else {
             if (any(k= is.na(match(x$left$stateid, names(statedata)))))
                 stop(x$left$stateid[k], ": state variable not found")
             zz <- statedata[[x$left$stateid]]
             if (any(k= is.na(match(x$left$value, zz))))
                 stop(x$left$value[k], ": state value not found")
             temp1 <- which(zz %in% x$left$value)
         }
     }
     else {
         k <- match(x$left, statedata$state)
         if (any(is.na(k))) stop(x$left[is.na(k)], ": state not found")
         temp1 <- which(statedata$state %in% x$left)
     }

     # right of colon
     if (!is.list(x$right) && length(x$right) ==1 && x$right ==0)
         temp2 <- 1:nrow(statedata)
     else if (is.numeric(x$right)) {
         temp2 <- as.integer(x$right)
         if (any(temp2 != x$right)) stop("non-integer state number")
         if (any(temp2 <1 | temp2> nstate))
             stop("numeric state is out of range")
     }
     else if (is.list(x$right) && names(x$right)[1] == "stateid") {
         if (is.null(x$right$value))
             stop("state variable with no list of values: ",x$right$stateid)
         else {
             if (any(k= is.na(match(x$right$stateid, names(statedata)))))
                 stop(x$right$stateid[k], ": state variable not found")
             zz <- statedata[[x$right$stateid]]
             if (any(k= is.na(match(x$right$value, zz))))
                 stop(x$right$value[k], ": state value not found")
             temp2 <- which(zz %in% x$right$value)
         }
     }
     else {
         k <- match(x$right, statedata$state)
         if (any(is.na(k))) stop(x$right[k], ": state not found")
         temp2 <- which(statedata$state %in% x$right)
     }


     state1 <- c(state1, rep(temp1, length(temp2)))
     state2 <- c(state2, rep(temp2, each=length(temp1)))
 }
```

At the end it has created to vectors state1 and state2 listing all the pairs of states that are indicated.

The init clause (initial values) are gathered but not checked: we don't yet know how many columns a term will expand into. tmap is a 3 way array: term, state1, state2 containing coefficient numbers and zeros.

⟨*parse-tmap*⟩=

```
 for (i in 1:length(covar1$rhs)) {
```

```
rhs <- covar1$rhs[[i]]
lhs <- covar1$lhs[[i]]  # one rhs and one lhs per formula

⟨parse-lmatch⟩
npair <- length(state1)  # number of state:state pairs for this line

# update tmap for this set of transitions
# first, what variables are mentioned, and check for errors
rterm <- terms(rhs$formula)
rindex <- 1L + termmatch(attr(rterm, "factors"), allterm)

# the update.formula function is good at identifying changes
# formulas that start with  "- x" have to be pasted on carefully
temp <- substring(deparse(rhs$formula, width.cutoff=500), 2)
if (substring(temp, 1,1) == ’-’) dummy <- formula(paste("~ .", temp))
else dummy <- formula(paste("~. +", temp))

rindex1 <- termmatch(attr(terms(dformula), "factors"), allterm)
rindex2 <- termmatch(attr(terms(update(dformula, dummy)), "factors"),
                 allterm)
dropped <- 1L + rindex1[is.na(match(rindex1, rindex2))] # remember the intercept
if (length(dropped) >0) {
    for (k in 1:npair) tmap[dropped, state1[k], state2[k]] <- 0
}

# grab initial values
if (length(rhs$ival))
    inits <- c(inits, list(term=rindex, state1=state1,
                           state2= state2, init= rhs$ival))

# adding -1 to the front is a trick, to check if there is a "+1" term
dummy <- ~ -1 + x
dummy[[2]][[3]] <- rhs$formula
if (attr(terms(dummy), "intercept") ==1) rindex <- c(1L, rindex)

# an update of "- sex" won’t generate anything to add
# dmap is simply an indexed set of unique values to pull from, so that
#  no number is used twice
if (length(rindex) > 0) {  # rindex = things to add
    if (rhs$common) {
        j <- dmap[rindex, state1[1], state2[1]]
        for(k in 1:npair) tmap[rindex, state1[k], state2[k]] <- j
    }
    else {
        for (k in 1:npair)
            tmap[rindex, state1[k], state2[k]] <- dmap[rindex, state1[k], state2[k]]
        }
    }

    # Deal with the shared argument, using - for a separate coef
    if (rhs$shared && npair>1) {
        j <- dmap[1, state1[1], state2[1]]
        for (k in 2:npair)
            tmap[1, state1[k], state2[k]] <- -j
    }
}
```

Fold the 3-dimensional tmap into a matrix with terms as rows and one column for each transition that actually occured.

⟨*parse-finish*⟩=

```
i <- match("(censored)", colnames(transitions), nomatch=0)
if (i==0) t2 <- transitions
else t2 <- transitions[,-i, drop=FALSE]   # transitions to ’censor’ don’t count
indx1 <- match(rownames(t2), states)
indx2 <- match(colnames(t2), states)
tmap2 <- matrix(0L, nrow= 1+nterm, ncol= sum(t2>0))

trow <- row(t2)[t2>0]
tcol <- col(t2)[t2>0]
for (i in 1:nrow(tmap2)) {
    for (j in 1:ncol(tmap2))
        tmap2[i,j] <- tmap[i, indx1[trow[j]], indx2[tcol[j]]]
}

# Remember which hazards had ph
# tmap2[1,] is the ’intercept’ row
# If the hazard for colum 6 is proportional to the hazard for column 2,
# the tmap2[1,2] = tmap[1,6], and phbaseline[6] =2
temp <- tmap2[1,]
indx <- which(temp> 0)
tmap2[1,] <- indx[match(abs(temp), temp[indx])]
phbaseline <- ifelse(temp<0, tmap2[1,], 0)    # remembers column numbers
tmap2[1,] <- match(tmap2[1,], unique(tmap2[1,])) # unique strata 1,2, ...

if (nrow(tmap2) > 1)
    tmap2[-1,] <- match(tmap2[-1,], unique(c(0L, tmap2[-1,]))) -1L

dimnames(tmap2) <- list(c("(Baseline)", colnames(allterm)),
                            paste(indx1[trow], indx2[tcol], sep=’:’))
# mapid gives the from,to for each realized state
list(tmap = tmap2, inits=inits, mapid= cbind(from=indx1[trow], to=indx2[tcol]),
     phbaseline = phbaseline)
```

Last is a helper routine that converts tmap, which has one row per term, into cmap, which has one row per coefficient. Both have one column per transition. It uses the assign attribute of the X matrix along with the column names.

Consider the model `x1 + strata(x2) + factor(x3)` where x3 has 4 levels. The Xassign vector will be 1, 3, 3, 3, since it refers to terms and there are 3 columns of X for term number 3. If there were an intercept the first column of X would be a 1 and Xassign would be 0, 1, 3, 3, 3.

Let's say that there were 3 transitions and tmap looks like this:

	1:2	1:3	2:3
(Baseline)	1	2	3
x1	1	4	4
strata(x2)	2	5	6
factor(x3)	3	3	7

The cmap matrix will ignore rows 1 and 3 since they do not correspond to coefficients in the model. Proportional baseline hazards add another wrinkle: say that the 1:3 and 2:3 hazards were proportional, and the user had `1:3 + 2:3 /shared` in thier call. Then the phbaseline vector will be 0,0,2 and cmap will gain an extra row with label ph(1:3) which has a coefficient for the 2:3 transition. If the user typed `2:3 + 1:3/shared` then the phbaseline vector will be (0,3,0) and 2:3 is the reference level.

⟨*parsecovar*⟩=
```
 parsecovar3 <- function(tmap, Xcol, Xassign, phbaseline=NULL) {
     # sometime X will have an intercept, sometimes not; cmap never does
     hasintercept <- (Xassign[1] ==0)
     ph.coef <- (phbaseline !=0)  # any proportional baselines?
     ph.rows <- length(unique(phbaseline[ph.coef])) #extra rows to add to cmap
     cmap <- matrix(0L, length(Xcol) + ph.rows -hasintercept, ncol(tmap))
     uterm <- unique(Xassign[Xassign != 0L])  # terms that will have coefficients

     xcount <- table(factor(Xassign, levels=1:max(Xassign)))
     mult <- 1L+ max(xcount)  # temporary scaling

     ii <- 0
     for (i in uterm) {
         k <- seq_len(xcount[i])
         for (j in 1:ncol(tmap))
             cmap[ii+k, j] <- if(tmap[i+1,j]==0) 0L else tmap[i+1,j]*mult +k
         ii <- ii + max(k)
     }

     if (ph.rows > 0) {
         temp <- phbaseline[ph.coef] # where each points
         for (i in unique(temp)) {
             # for each baseline that forms a reference
             j <- which(phbaseline ==i)  # the others that are proportional to it
             k <- seq_len(length(j))
             ii <- ii +1   # row of cmat for this baseline
             cmap[ii, j] <- max(cmap) + k  # fill in elements
```

```
        }
        newname <- paste0("ph(", colnames(tmap)[unique(temp)], ")")
    } else newname <- NULL

    # renumber coefs as 1, 2, 3, ...
    cmap[,] <- match(cmap, sort(unique(c(0L, cmap)))) -1L

    colnames(cmap) <- colnames(tmap)
    if (hasintercept) rownames(cmap) <- c(Xcol[-1], newname)
    else rownames(cmap) <- c(Xcol, newname)

    cmap
}
```

6 Person years

The person years routine and the expected survival code are the two parts of the survival package that make use of external rate tables, of which the United States mortality tables `survexp.us` and `survexp.usr` are examples contained in the package. The arguments for pyears are

formula The model formula. The right hand side consists of grouping variables and is essentially identical to [[survfit]], the result of the model will be a table of results with dimensions determined from the right hand variables. The formula can include an optional [[ratetable]] directive; but this style has been superseded by the [[rmap]] argument.

data, weights, subset, na.action as usual

rmap an optional mapping for rate table variables, see more below.

ratetable the population rate table to use as a reference. This can either be a ratetable object or a previously fitted Cox model

scale Scale the resulting output times, e.g., 365.25 to turn days into years.

expect Should the output table include the expected number of events, or the expected number of person-years of observation?

model, x, y as usual

data.frame if true the result is returned as a data frame, if false as a set of tables.

```
⟨pyears⟩=
pyears <- function(formula, data,
        weights, subset, na.action, rmap,
        ratetable, scale=365.25,  expect=c('event', 'pyears'),
        model=FALSE, x=FALSE, y=FALSE, data.frame=FALSE) {

    ⟨pyears-setup⟩
    ⟨pyears-compute⟩
    ⟨pyears-finish⟩
    }
```

Start out with the standard model processing, which involves making a copy of the input call, but keeping only the arguments we want. We then process the special argument `rmap`. This is discussed in the section on the `survexp` function so we need not repeat the explantation here.

```
⟨pyears-setup⟩=
expect <- match.arg(expect)
Call <- match.call()

# create a call to model.frame() that contains the formula (required)
#  and any other of the relevant optional arguments
# then evaluate it in the proper frame
indx <- match(c("formula", "data", "weights", "subset", "na.action"),
                  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

Terms <- if(missing(data)) terms(formula, ’ratetable’)
         else              terms(formula, ’ratetable’,data=data)
if (any(attr(Terms, ’order’) >1))
        stop("Pyears cannot have interaction terms")

rate <- attr(Terms, "specials")$ratetable
if (length(rate) >0 || !missing(rmap) || !missing(ratetable)) {
    has.ratetable <- TRUE
    if(length(rate) > 1)
        stop("Can have only 1 ratetable() call in a formula")
    if (missing(ratetable)) stop("No rate table specified")

    ⟨survexp-setup-rmap⟩
    }
else has.ratetable <- FALSE

mf <- eval(tform, parent.frame())

Y <- model.extract(mf, ’response’)
if (is.null(Y)) stop ("Follow-up time must appear in the formula")
if (!is.Surv(Y)){
    if (any(Y <0)) stop ("Negative follow up time")
    Y <- as.matrix(Y)
    if (ncol(Y) >2) stop("Y has too many columns")
    }
else {
    stype <- attr(Y, 'type')
    if (stype == 'right') {
        if (any(Y[,1] <0)) stop("Negative survival time")
        nzero <- sum(Y[,1]==0 & Y[,2] ==1)
        if (nzero >0)
            warning(paste(nzero,
                     "observations with an event and 0 follow-up time,",
                   "any rate calculations are statistically questionable"))
        }
    else if (stype != 'counting')
        stop("Only right-censored and counting process survival types are supported")
    }

n <- nrow(Y)
if (is.null(n) || n==0) stop("Data set has 0 observations")

weights <- model.extract(mf, 'weights')
if (is.null(weights)) weights <- rep(1.0, n)
```

The next step is to check out the ratetable. For a population rate table a set of consistency checks is done by the `match.ratetable` function, giving a set of sanitized indices `R`. This function wants characters turned to factors. For a Cox model `R` will be a model matix whose covariates are coded in exactly the same way that variables were coded in the original Cox model. We call the model.matrix.coxph function so as not to have to repeat the steps found there (remove cluster statements, etc).

⟨*pyears-setup*⟩=

```
# rdata contains the variables matching the ratetable
if (has.ratetable) {
    rdata <- data.frame(eval(rcall, mf), stringsAsFactors=TRUE)
    if (is.ratetable(ratetable)) {
        israte <- TRUE
        rtemp <- match.ratetable(rdata, ratetable)
        R <- rtemp$R
        }
    else if (inherits(ratetable, 'coxph') && !inherits(ratetable, "coxphms")) {
        israte <- FALSE
        Terms <- ratetable$terms
        if (!is.null(attr(Terms, 'offset')))
            stop("Cannot deal with models that contain an offset")
        strats <- attr(Terms, "specials")$strata
        if (length(strats))
            stop("pyears cannot handle stratified Cox models")

        if (any(names(mf[,rate]) !=  attr(ratetable$terms, 'term.labels')))
             stop("Unable to match new data to old formula")
        R <- model.matrix.coxph(ratetable, data=rdata)
        }
    else stop("Invalid ratetable")
    }
```

Now we process the non-ratetable variables. Those of class `tcut` set up time-dependent classes. For these the cutpoints attribute sets the intervals, if there were 4 cutpoints of 1, 5,6, and 10 the 3 intervals will be 1-5, 5-6 and 6-10, and odims will be 3. All other variables are treated as factors.

⟨*pyears-setup*⟩=
```
ovars <- attr(Terms, 'term.labels')
if (length(ovars)==0)  {
    # no categories!
    X <- rep(1,n)
    ofac <- odim <- odims <- ocut <- 1
    }
else {
    odim <- length(ovars)
    ocut <- NULL
    odims <- ofac <- double(odim)
    X <- matrix(0, n, odim)
    outdname <- vector("list", odim)
    names(outdname) <- attr(Terms, 'term.labels')
    for (i in 1:odim) {
        temp <- mf[[ovars[i]]]
        if (inherits(temp, 'tcut')) {
            X[,i] <- temp
            temp2 <- attr(temp, 'cutpoints')
            odims[i] <- length(temp2) -1
            ocut <- c(ocut, temp2)
            ofac[i] <- 0
            outdname[[i]] <- attr(temp, 'labels')
            }
        else {
            temp2 <- as.factor(temp)
            X[,i] <- temp2
            temp3 <- levels(temp2)
            odims[i] <- length(temp3)
            ofac[i] <- 1
            outdname[[i]] <- temp3
            }
    }
}
```

Now do the computations. The code above has separated out the variables into 3 groups:

- The variables in the rate table. These determine where we *start* in the rate table with respect to retrieving the relevant death rates. For the US table [[survexp.us]] this will be the date of study entry, age (in days) at study entry, and sex of each subject.

- The variables on the right hand side of the model. These are interpreted almost identically to a call to [[table]], with special treatment for those of class *tcut*.
- The response variable, which tells the number of days of follow-up and optionally the status at the end of follow-up.

Start with the rate table variables. There is an oddity about US rate tables: the entry for age (year=1970, age=55) contains the daily rate for anyone who turns 55 in that year, from their birthday forward for 365 days. So if your birthday is on Oct 2, the 1970 table applies from 2Oct 1970 to 1Oct 1971. The underlying C code wants to make the 1970 rate table apply from 1Jan 1970 to 31Dec 1970. The easiest way to finess this is to fudge everyone's enter-the-study date. If you were born in March but entered in April, make it look like you entered in Febuary; that way you get the first 11 months at the entry year's rates, etc. The birth date is entry date - age in days (based on 1/1/1970).

The other aspect of the rate tables is that "older style" tables, those that have the factor attribute, contained only decennial data which the C code would interpolate on the fly. The value of `atts$factor` was 10 indicating that there are 10 years in the interpolation interval. The newer tables do not do this and the C code is passed a 0/1 for continuous (age and year) versus discrete (sex, race).

```
⟨pyears-compute⟩=
 ocut <-c(ocut,0)   #just in case it were of length 0
 osize <- prod(odims)
 if (has.ratetable) {  #include expected
     atts <- attributes(ratetable)
     datecheck <- function(x)
         inherits(x, c("Date", "POSIXt", "date", "chron"))
     cuts <- lapply(attr(ratetable, "cutpoints"), function(x)
         if (!is.null(x) & datecheck(x)) ratetableDate(x) else x)

     if (is.null(atts$type)) {
         #old stlye table
         rfac <- atts$factor
         us.special <- (rfac >1)
         }
     else {
         rfac <- 1*(atts$type ==1)
         us.special <- (atts$type==4)
         }
     if (any(us.special)) {  #special handling for US pop tables
         if (sum(us.special) > 1) stop("more than one type=4 in a rate table")
         # Someone born in June of 1945, say, gets the 1945 US rate until their
         #  next birthday.  But the underlying logic of the code would change
         #  them to the 1946 rate on 1/1/1946, which is the cutpoint in the
```

```
    #  rate table.  We fudge by faking their enrollment date back to their
    #  birth date.
    #
    # The cutpoint for year has been converted to days since 1/1/1970 by
    #  the ratetableDate function.  (Date objects in R didn't exist when
    #  rate tables were conceived.)
    if (is.null(atts$dimid)) dimid <- names(atts$dimnames)
    else dimid <- atts$dimid
    cols <- match(c("age", "year"), dimid)
    if (any(is.na(cols)))
        stop("ratetable does not have expected shape")

    # The format command works for Dates, use it to get an offset
    bdate <- as.Date("1970-01-01") + (R[,cols[2]] - R[,cols[1]])
    byear <- format(bdate, "%Y")
    offset <- as.numeric(bdate - as.Date(paste0(byear, "-01-01")))
    R[,cols[2]] <- R[,cols[2]] - offset

    # Doctor up "cutpoints" - only needed for (very) old style rate tables
    #  for which the C code does interpolation on the fly
    if (any(rfac >1)) {
        temp <-  which(us.special)
        nyear <- length(cuts[[temp]])
        nint <- rfac[temp]       #intervals to interpolate over
        cuts[[temp]] <- round(approx(nint*(1:nyear), cuts[[temp]],
                                nint:(nint*nyear))$y - .0001)
        }
    }
docount <- is.Surv(Y)
temp <- .C(Cpyears1,
                as.integer(n),
                as.integer(ncol(Y)),
                as.integer(is.Surv(Y)),
                as.double(Y),
                as.double(weights),
                as.integer(length(atts$dim)),
                as.integer(rfac),
                as.integer(atts$dim),
                as.double(unlist(cuts)),
                as.double(ratetable),
                as.double(R),
                as.integer(odim),
                as.integer(ofac),
                as.integer(odims),
                as.double(ocut),
                as.integer(expect=='event'),
                    as.double(X),
                    pyears=double(osize),
                    pn    =double(osize),
                    pcount=double(if(docount) osize else 1),
                    pexpect=double(osize),
                    offtable=double(1))[18:22]
    }
else {   #no expected
    docount <- as.integer(ncol(Y) >1)
    temp <- .C(Cpyears2,
                    as.integer(n),
                    as.integer(ncol(Y)),
                    as.integer(docount),
                    as.double(Y),
                    as.double(weights),
                    as.integer(odim),
                    as.integer(ofac),
                    as.integer(odims),
                    as.double(ocut),
                    as.double(X),
                    pyears=double(osize),
                    pn    =double(osize),
                    pcount=double(if (docount) osize else 1),
                    offtable=double(1)) [11:14]
    }
```

Create the output object.

```
⟨pyears-finish⟩=
has.tcut <- any(sapply(mf, function(x) inherits(x, 'tcut')))
if (data.frame) {
    # Create a data frame as the output, rather than a set of
    #  rate tables
    if (length(ovars) ==0) {  # no variables on the right hand side
        keep <- TRUE
        df <- data.frame(pyears= temp$pyears/scale,
                         n = temp$n)
    }
    else {
        keep <- (temp$pyears >0)  # what rows to keep in the output
        # grab prototype rows from the model frame, this preserves class
        #  (unless it is a tcut variable, then we know what to do)
        tdata <- lapply(1:length(ovars), function(i) {
            temp <- mf[[ovars[i]]]
            if (inherits(temp, "tcut")) { #if levels are numeric, return numeric
                if (is.numeric(outdname[[i]])) outdname[[i]]
                else  factor(outdname[[i]], outdname[[i]]) # else factor
            }
            else temp[match(outdname[[i]], temp)]
        })
        tdata$stringsAsFactors <- FALSE  # argument for expand.grid
        df <- do.call("expand.grid", tdata)[keep,,drop=FALSE]
        names(df) <- ovars
        df$pyears <- temp$pyears[keep]/scale
        df$n <- temp$pn[keep]
    }
    row.names(df) <- NULL   # toss useless 'creation history'
    if (has.ratetable) df$expected <- temp$pexpect[keep]
    if (expect=='pyears') df$expected <- df$expected/scale
    if (docount) df$event <- temp$pcount[keep]
    # if any of the predictors were factors, make them factors in the output
    for (i in 1:length(ovars)){
        if (is.factor( mf[[ovars[i]]]))
            df[[ovars[i]]] <- factor(df[[ovars[i]]], levels( mf[[ovars[i]]]))
    }

    out <- list(call=Call,
                data= df, offtable=temp$offtable/scale,
                tcut=has.tcut)
    if (has.ratetable && !is.null(rtemp$summ))
        out$summary <- rtemp$summ
}

else if (prod(odims) ==1) {  #don't make it an array
    out <- list(call=Call, pyears=temp$pyears/scale, n=temp$pn,
                offtable=temp$offtable/scale, tcut = has.tcut)
    if (has.ratetable) {
        out$expected <- temp$pexpect
        if (expect=='pyears') out$expected <- out$expected/scale
        if (!is.null(rtemp$summ)) out$summary <- rtemp$summ
    }
    if (docount) out$event <- temp$pcount
}
else {
    out <- list(call = Call,
            pyears= array(temp$pyears/scale, dim=odims, dimnames=outdname),
            n     = array(temp$pn,     dim=odims, dimnames=outdname),
            offtable = temp$offtable/scale, tcut=has.tcut)
    if (has.ratetable) {
        out$expected <- array(temp$pexpect, dim=odims, dimnames=outdname)
        if (expect=='pyears') out$expected <- out$expected/scale
        if (!is.null(rtemp$summ)) out$summary <- rtemp$summ
    }
    if (docount)
            out$event <- array(temp$pcount, dim=odims, dimnames=outdname)
}
out$observations <- nrow(mf)
out$terms <- Terms
na.action <- attr(mf, "na.action")
if (length(na.action))  out$na.action <- na.action
if (model) out$model <- mf
else {
    if (x) out$x <- X
    if (y) out$y <- Y
}
class(out) <- 'pyears'
out
```

6.1 Print and summary

The print function for pyear gives a very abbreviated printout: just a few lines. It works with pyears objects with or without a data component.

```
⟨print.pyears⟩=
print.pyears <- function(x, ...) {
    if (!is.null(cl<- x$call)) {
        cat("Call:\n")
        dput(cl)
        cat("\n")
        }

    if (is.null(x$data)) {
        if (!is.null(x$event))
            cat("Total number of events:", format(sum(x$event)), "\n")
        cat (   "Total number of person-years tabulated:",
             format(sum(x$pyears)),
             "\nTotal number of person-years off table:",
             format(x$offtable), "\n")
        }
    else {
        if (!is.null(x$data$event))
            cat("Total number of events:", format(sum(x$data$event)), "\n")
        cat (   "Total number of person-years tabulated:",
             format(sum(x$data$pyears)),
             "\nTotal number of person-years off table:",
             format(x$offtable), "\n")
        }
    if (!is.null(x$summary)) {
        cat("Matches to the chosen rate table:\n  ",
            x$summary)
        }
    cat("Observations in the data set:", x$observations, "\n")
    if (!is.null(x$na.action))
      cat("  (", naprint(x$na.action), ")\n", sep='')
    cat("\n")
    invisible(x)
}
```

The summary function attempts to create output that looks like a pandoc table, which in turn makes it mesh nicely with Rstudio. Pandoc has 4 types of tables: with and without vertical bars and with single or multiple rows per cell. If the pyears object has only a single dimension then our output will be a simple table with a row or column for each of the output types (see the vertical argument). The result will be a simple table or a “pipe” table depending on the vline argument. For two or more dimensions the output follows the usual R strategy for printing an array, but with each “cell” containing all of the summaries for that combination of predictors, thus giving either a “multiline” or “grid” table. The default values of no vertical lines makes the tables appropriate for non-pandoc output such as a terminal session.

⟨*print.pyears*⟩=

```
summary.pyears <- function(object, header=TRUE, call=header,
                           n= TRUE, event=TRUE, pyears=TRUE,
                           expected = TRUE, rate = FALSE, rr = expected,
                           ci.r = FALSE, ci.rr = FALSE, totals=FALSE,
                           legend=TRUE, vline = FALSE, vertical = TRUE,
                           nastring=".", conf.level=0.95,
                           scale= 1, ...) {
    # Usual checks
    if (!inherits(object, "pyears"))
        stop("input must be a pyears object")
    temp <- c(is.logical(header), is.logical(call), is.logical(n),
              is.logical(event) , is.logical(pyears), is.logical(expected),
              is.logical(rate), is.logical(ci.r), is.logical(rr),
              is.logical(ci.rr), is.logical(vline), is.logical(vertical),
              is.logical(legend), is.logical(totals))
    tname <- c("header", "call", "n", "event", "pyears", "expected",
               "rate", "ci.r", "rr", "ci.rr", "vline", "vertical",
               "legend", "totals")
    if (any(!temp) || length(temp) != 14 || any(is.na(temp))) {
        stop("the ", paste(tname[!temp], collapse=", "),
             "argument(s) must be single logical values")
    }
    if (!is.numeric(conf.level) || conf.level <=0 || conf.level >=1 |
        length(conf.level) > 1 || is.na(conf.level) > 1)
        stop("conf.level must be a single numeric between 0 and 1")
    if (is.na(scale) || !is.numeric(scale) || length(scale) !=1 || scale <=0)
        stop("scale must be a value > 0")

    vname <- attr(terms(object), "term.labels")  #variable names

    if (!is.null(object$data)) {
        # Extra work: restore the tables which had been unpacked into a df
        #  All of the categories are factors in this case
        tdata <- object$data[vname]  # the conditioning variables
        dname <- lapply(tdata, function(x) {
            if (is.factor(x)) levels(x) else sort(unique(x))}) # dimnames
        dd  <-   sapply(dname, length)                # dim of arrays
        index <- tapply(tdata[,1], tdata)
        restore <- c(’n’, ’event’, ’pyears’, ’expected’) #do these, if present
        restore <- restore[restore %in% names(object$data)]
        new   <- lapply(object$data[restore],
                        function(x) {
                            temp <- array(0L, dim=dd, dimnames=dname)
                            temp[index] <- x
                            temp} )
        object <- c(object, new)
    }

    if (is.null(object$expected)) {
        expected <- FALSE
        rr <- FALSE
        ci.rr <- FALSE
    }
    if (is.null(object$event)) {
        event <- FALSE
        rate <- FALSE
        ci.r <- FALSE
        rr <- FALSE
        ci.rr <- FALSE
    }

    # print out the front matter
    if (call && !is.null(object$call)) {
        cat("Call: ")
        dput(object$call)
        cat("\n")
    }
    if (header) {
        cat("number of observations =", object$observations)
        if (length(object$omit))
            cat("  (", naprint(object$omit), ")\n", sep="")
        else cat("\n")
```

```
        if (object$offtable > 0)
            cat(" Total time lost (off table)", format(object$offtable), "\n")
        cat("\n")
    }

    # Add in totals if requested
    if (totals) {
        # if the pyear object was based on any time dependent cuts, then
        #  the "n" component cannot be totaled up.
        tcut <- if (is.null(object$tcut)) TRUE else object$tcut
        object$n <- pytot(object$n, na=tcut)
        object$pyears <- pytot(object$pyears)
        if (event) object$event <- pytot(object$event)
        if (expected) object$expected <- pytot(object$expected)
    }

    dd <- dim(object$n)
    vname <- attr(terms(object), "term.labels")  #variable names
    ⟨pyears-list⟩
    if (length(dd) ==1) {
        # 1 dimensional table
        ⟨pyears-table1⟩
    } else {
        # more than 1 dimension
        ⟨pyears-table2⟩
    }
    invisible(object)
}

⟨pyears-charfun⟩
```

```
⟨pyears-list⟩=
# Put the elements to be printed onto a list
pname <- (tname[3:6])[c(n, event, pyears, expected)]
plist <- object[pname]

if (rate) {
    pname <- c(pname, "rate")
    plist$r <- scale* object$event/object$pyears
}
if (ci.r) {
    pname <- c(pname, "ci.r")
    plist$ci.r <- cipoisson(object$event, object$pyears, p=conf.level) *scale
}
if (rr) {
    pname <- c(pname, "rr")
    plist$rr <- object$event/object$expected
}
if (ci.rr) {
    pname <- c(pname, "ci.rr")
    plist$ci.rr <-  cipoisson(object$event, object$expected, p=conf.level)
}

rname <- c(n = "N", event="Events",
           pyears= "Time", expected= "Expected events",
           rate = "Event rate", ci.r = "CI (rate)",
           rr= "Obs/Exp",   ci.rr= "CI (O/E)")
rname <- rname[pname]
```

If there is only one dimension to the table we can forgo the top legend and use the object names as one of the margins. If `vertical=TRUE` the output types are vertical, otherwise they are horizontal. Format each element of the output separately.

⟨*pyears-table1*⟩=

```
cname <- names(object$n)  #category names

if (vertical) {
    # The person-years objects list across the top, categories up and down
    # This makes columns line up in a standard "R" way
    # The first column label is the variable name, content is the categories
    plist <- lapply(plist, pformat, nastring, ...) # make it character
    pcol  <- sapply(plist, function(x) nchar(x[1])) #width of each one
    colwidth <- pmax(pcol, nchar(rname)) +2
    for (i in 1:length(plist))
        plist[[i]] <- strpad(plist[[i]], colwidth[i])

    colwidth <- c(max(nchar(vname), nchar(cname)) +2, colwidth)
    leftcol <- list(strpad(cname, colwidth[1]))
    header  <- strpad(c(vname, rname), colwidth)
}
else {
    # in this case each column will have different types of objects in it
    #  alignment is the nuisance
    newmat <- pybox(plist, length(plist[[1]]), nastring, ...)
    colwidth <- pmax(nchar(cname), apply(nchar(newmat), 1, max)) +2
    # turn the list sideways
    plist <- split(newmat, row(newmat))
    for (i in 1:length(plist))
        plist[[i]] <- strpad(plist[[i]], colwidth[i])

    colwidth <- c(max(nchar(vname), nchar(rname)) +2, colwidth)
    leftcol <- list(strpad(rname, colwidth[1]))
    header  <- strpad(c(vname, cname), colwidth)
 }

# Now print it
if (vline) { # use a pipe table
    cat(paste(header, collapse = "|"), "\n")
    cat(paste(strpad("-", colwidth, "-"), collapse="|"), "\n")

    temp <- do.call("paste", c(leftcol, plist, list(sep ="|")))
    cat(temp, sep= ’\n’)
}
else {
    cat(paste(header, collapse = " "), "\n")
    cat(paste(strpad("-", colwidth, "-"), collapse=" "), "\n")
    temp <- do.call("paste", c(leftcol, plist, list(sep =" ")))
    cat(temp, sep=’\n’)
}
```

When there are more than one category in the pyears object then we use a special layout. Each ’cell’ of the printed table has all of the values in it.

⟨*pyears-table2*⟩=

```
if (header) {
    # the header is itself a table
    width <- max(nchar(rname))
    if (vline) {
        cat(’+’, strpad(’-’, width, ’-’), "+\n", sep="")
        cat(paste0(’|’,strpad(rname, width), ’|’), sep=’\n’)
        cat(’+’, strpad(’-’, width, ’-’), "+\n\n", sep="")
    } else {
        cat(strpad(’-’, width, ’-’), "\n")
        cat(strpad(rname, width), sep=’\n’)
        cat(strpad(’-’, width, ’-’), "\n\n")
    }
}
tname <- vname[1:2]  #names for the row and col
rowname  <- dimnames(object$n)[[1]]
colname  <- dimnames(object$n)[[2]]
if (length(dd) > 2)
    newmat <- pybox(plist, c(dd[1],dd[2], prod(dd[-(1:2)])),
                    nastring, ...)
else  newmat <- pybox(plist, dd,  nastring, ...)

if (length(dd) > 2) {
    newmat <- pybox(plist, c(dd[1],dd[2], prod(dd[-(1:2)])),
                    nastring, ...)
    outer.label <- do.call("expand.grid", dimnames(object$n)[-(1:2)])
    temp <- names(outer.label)
    for (i in 1:nrow(outer.label)) {
        # first the caption, then data
        cat(paste(":", paste(temp, outer.label[i,], sep="=")), '\n')
        pyshow(newmat[,,i,], tname, rowname, colname, vline)
    }
}
else {
    newmat <- pybox(plist, dd,  nastring, ...)
    pyshow(newmat, tname, rowname, colname, vline)
}
```

Here are some character manipulation functions. The stringi package has more elegant versions of the pad function, but we don't need the speed. No one is going to print out thousands of lines.

⟨*pyears-charfun*⟩=

```
strpad <- function(x, width, pad=' ') {
    # x = the string(s) to be padded out
    # width = width of desired string.
    nc <- nchar(x)
    added <- width - nc

    left  <- pmax(0, floor(added/2))       # can't add negative space
    right <- pmax(0, width - (nc + left))  # right will be >= left

    if (all(right <=0)) {
        if (length(x) >= length(width)) x  # nothing needs to be done
        else rep(x, length=length(width))
    }
    else {
        # Each pad could be a different length.
        # Make a long string from which we can take a portion
        longpad <- paste(rep(pad, max(right)), collapse='')
        paste0(substring(longpad, 1, left), x, substring(longpad,1, right))
    }
}

pformat <- function(x, nastring, ...) {
    # This is only called for single index tables, in vertical mode
    # Any matrix will be a confidence interval
    if (is.matrix(x))
        ret <- paste(ifelse(is.na(x[,1]), nastring,
                            format(x[,1],  ...)), "-",
                     ifelse(is.na(x[,2]), nastring,
                            format(x[,2],  ...)))
    else ret <- ifelse(is.na(x), nastring, format(x,  ...))
}
```

Create formatted boxes. We want all the decimal points to line up, so the format calls are in 3 parts: integer, real, and confidence interval. If there are confidence intervals, format their values and then paste together the left-right ends. The intermediag form `final` is a matrix with one column per statistic. At the end, reformat it as an array whose last dimension is the components.

⟨*pyears-charfun*⟩=

```
 pybox <- function(plist, dd, nastring, ...) {
     ci <- (substring(names(plist), 1,3) == "ci.")  # the CI components
     int <- sapply(plist, function(x) all(x == floor(x) | is.na(x)))
     int <- (!ci & int)
     real<- (!ci & !int)
     nc <- prod(dd)
     final <- matrix("", nrow=nc, ncol=length(ci))

     if (any(int)) { # integers
         if (any(sapply(plist[int], length) != nc))
             stop("programming length error, notify package author")
         temp <- unlist(plist[int])
         final[,int] <- ifelse(is.na(temp), nastring, format(temp))
     }
     if (any(real)) { # floating point
         if (any(sapply(plist[real], length) != nc))
             stop("programming length error, notify package author")
         temp <- unlist(plist[real])
         final[,real] <- ifelse(is.na(temp), nastring,
                                format(temp,  ...))
     }

     if (any(ci)) {
         if (any(sapply(plist[ci], length) != nc*2))
             stop("programming length error, notify package author")
         temp <- unlist(plist[ci])
         temp <- array(ifelse(is.na(temp), nastring,
                              format(temp,  ...)),
                       dim=c(nc, 2, sum(ci)))
         final[,ci] <- paste(temp[,1,], temp[,2,], sep='-')
     }
     array(final, dim=c(dd, length(ci)))
 }
```

This function prints out a box table. Each cell contains the full set of statistics that were requested. Most of the work is the creation of the appropriate spacing and special characters to create a valid pandoc table.

⟨*pyears-charfun*⟩=
```
 pyshow <- function(dmat, labels, rowname, colname, vline) {
     # Every column is the same width, except the first
     colwidth <- c(max(nchar(rowname), nchar(labels[1])),
                   rep(max(nchar(dmat[1,1,]), nchar(colname)), length(colname)))
     colwidth[2] <- max(colwidth[2], nchar(labels[2]))
     ncol <- length(colwidth)

     dd <- dim(dmat)  # vector of length 3, third dim is the statistics
     rline <- ceiling(dd[3]/2)  #which line to put the row label on.
     if (vline) { # use a grid table
         cat("+", paste(strpad(’-’, colwidth, pad=’-’), collapse=’+’), "+\n",
             sep=’’)
         temp <- rep(’ ’, ncol); temp[2] <- labels[2]
         cat("|", paste(strpad(temp, colwidth), collapse="|"), "|\n",
             sep=’’)
         cat("|", paste(strpad(c(labels[1], colname), colwidth), collapse="|"),
             "|\n", sep=’’)
         cat("+", paste(strpad(’=’, colwidth, pad=’=’), collapse="+"), "+\n",
             sep=’’)
         for (i in 1:dd[1]) {
             for (j in 1:dd[3]) { #one printout line per stat
                 if (j==rline) temp <- c(rowname[i], dmat[i,,j])
                 else temp <- c("", dmat[i,,j])
                 cat("|", paste(strpad(temp, colwidth), collapse=’|’), "|\n",
                     sep=’’)
             }
             cat("+", paste(strpad(’-’, colwidth, ’-’), collapse=’+’), "+\n",
                 sep=’’)
         }
     }
     else { # use a multiline table
         cat(paste(strpad(’-’, colwidth, ’-’), collapse=’-’), "\n")
         temp <- rep(’ ’, ncol); temp[2] <- labels[2]
         cat(paste(strpad(temp, colwidth), collapse=" "), "\n")
         cat(paste(strpad(c(labels[1], colname), colwidth), collapse=" "),
             "\n")
         cat(paste(strpad(’-’, colwidth, pad=’-’), collapse=" "), "\n")
         for (i in 1:dd[1]) {
             for (j in 1:dd[3]) { #one printout line per stat
                 if (j==rline) temp <- c(rowname[i], dmat[i,,j])
                 else temp <- c("", dmat[i,,j])
                 cat(paste(strpad(temp, colwidth), collapse=’ ’), "\n")
             }
```

```
            if (i< dd[1]) cat(" \n") #blank line
        }
        cat(paste(strpad(’-’, colwidth, ’-’), collapse=’-’), "\n")
    }
}
```

This function adds a totals row to the data, for either the first or first and second dimensions. The “n” component can’t be totaled, so we turn that into NA.

⟨*pyears-charfun*⟩=

```
pytot <- function(x, na=FALSE) {
    dd <- dim(x)
    if (length(dd) ==1) {
        if (na) array(c(x, NA), dim= length(x) +1,
                              dimnames=list(c(dimnames(x)[[1]], "Total")))
        else array(c(x, sum(x)), dim= length(x) +1,
                              dimnames=list(c(dimnames(x)[[1]], "Total")))
    }
    else if (length(dd) ==2) {
        if (na) new <- rbind(cbind(x, NA), NA)
        else {
            new <- rbind(x, colSums(x))
            new <- cbind(new, rowSums(new))
            }
        array(new, dim=dim(x) + c(1,1),
              dimnames=list(c(dimnames(x)[[1]], "Total"),
                            c(dimnames(x)[[2]], "Total")))
    }
    else {
        # The general case
        index <- 1:length(dd)
        if (na) sum1 <- sum2 <- sum3 <- NA
        else {
            sum1 <- apply(x, index[-1], sum)    # row sums
            sum2 <- apply(x, index[-2], sum)    # col sums
            sum3 <- apply(x, index[-(1:2)], sum) # total sums
            }

        # create a new matrix and then fill it in
        d2 <- dd
        d2[1:2] <- dd[1:2] +1
        dname <- dimnames(x)
        dname[[1]] <- c(dname[[1]], "Total")
        dname[[2]] <- c(dname[[2]], "Total")
        new <- array(x[1], dim=d2, dimnames=dname)

        # say dim(x) =(5,8,4); we want new[6,-9,] <- sum1; new[-6,9,] <- sum2
        #  and new[6,9,] <- sum3
        # if dim is longer, we need to add more commas
        commas <- rep(',', length(dd) -2)
        eval(parse(text=paste("new[1:dd[1], 1:dd[2]", commas, "] <- x")))
        eval(parse(text=paste("new[ d2[1],-d2[2]", commas, "] <- sum1")))
        eval(parse(text=paste("new[-d2[1], d2[2]", commas, "] <- sum2")))
        eval(parse(text=paste("new[ d2[1], d2[2]", commas, "] <- sum3")))
        new
    }
}
```

7 Residuals for survival curves

7.1 R-code

For all the more complex cases, the variance of a survival curve is based on the infinitesimal jackknife:

$$D_i(t) = \frac{\partial S(t)}{\partial w_i}$$

evaluated at the the observed vector of weights. The variance at a given time is then $D'WD'$ where D is a diagonal matrix of the case weights. When there are multiple states S is replaced by the vector $p(t)$, with one element per state, and the formula gets a bit more complex. The predicted curve from a Cox model is the most complex case.

Realizing that we need to return the matrix D to the user, in order to compute the variance of derived quantities like the restricted mean time in state, the code has been changed from a primarily internal focus (compute within the survfit routine) to an external one.

The underlying C code is very similar to that in survfitkm.c One major difference in the routines is that this code is designed to return values at a fixed set of time points; it is an error if the user does not provide them. This allows the result to be presented as a matrix or array. Computational differences will be discussed later.

The method argument is for debugging. For multi-state it uses either C code or the optimized R method. The double call below is because we want residuals to return a simple matrix, but the pseudo function needs to get back a little bit more.

⟨*residuals.survfit*⟩=
```
# residuals for a survfit object
residuals.survfit <- function(object, times,
                              type= "pstate",
                              collapse, weighted=FALSE, method=1, ...){

    if (!inherits(object, "survfit"))
        stop("argument must be a survfit object")
    if (object$type=="interval") {
        # trial code to support it
        # reconstruct the data set
        # create dummy time/status for all interval or left censored
        #   over the span of jump points in S, non-censored obs with
        #   weights proportional to the jumps
        # combine dummy + (exact, right) from original, compute KM
        # get pseudo for this new KM
        # collapse dummy obs back to a single
        stop("residuals for interval-censored data are not available")
        }

    if (missing(times)) stop("the times argument is required")
    # allow a set of alias
    temp <- c("pstate", "cumhaz", "sojourn", "survival",
                              "chaz", "rmst", "rmts", "auc")
    type <- match.arg(casefold(type), temp)
    itemp <-  c(1,2,3,1,2,3,3,3)[match(type, temp)]
    type <- c("pstate", "cumhaz", "auc")[itemp]

    if (missing(collapse))
         fit <- survresid.fit(object, times, type, weighted=weighted,
                              method= method)
    else fit <- survresid.fit(object, times, type, collapse= collapse,
                              weighted= weighted, method= method)

    fit$residuals
}

survresid.fit <- function(object, times,
                              type= "pstate",
                              collapse, weighted=FALSE, method=1) {
    if (object$type=="interval") stop("interval censored not yet supported")
    survfitms <- inherits(object, "survfitms")
    coxsurv <- inherits(object, "survfitcox")
    timefix <- (is.null(object$timefix) || object$timefix)

    start.time <- object$start.time
    if (is.null(start.time)) start.time <- min(c(0, object$time))

    # check input arguments
    if (missing(times))
        stop ("the times argument is required")
    else {
        if (!is.numeric(times)) stop("times must be a numeric vector")
        times <- sort(unique(times))
        if (timefix) times <- aeqSurv(Surv(times))[,1]
    }
```

```
 # get the data
⟨rsurvfit-data⟩

 if (missing(collapse)) collapse <- (!(is.null(id)) && any(duplicated(id)))
 if (collapse && is.null(id)) stop("collapse argument requires an id or cluster argument in the

 ny <- ncol(newY)
 if (collapse && any(X != X[1])) {
     # If the same id shows up in multiple curves, we just can't deal
     #  with it.
     temp <- unlist(lapply(split(id, X), unique))
     if (any(duplicated(temp)))
         stop("same id appears in multiple curves, cannot collapse")
 }

 timelab <- signif(times, 3)  # used for dimnames
 # What type of survival curve?
 if (!coxsurv) {
     stype <- Call$stype
     if (is.null(stype)) stype <- 1
     ctype <- Call$ctype
     if (is.null(ctype)) ctype <- 1
     if (!survfitms) {
         resid <- rsurvpart1(newY, X, casewt, times,
                             type, stype, ctype, object)
         if (collapse) {
             resid <- rowsum(resid, id, reorder=FALSE)
             dimnames(resid) <- list(id= unique(id), times=timelab)
             curve <- (as.integer(X))[!duplicated(id)] #which curve for each
         }
         else {
             if (length(id) >0) dimnames(resid) <- list(id=id, times=timelab)
             curve <- as.integer(X)
         }
     }
     else {  # multi-state
         if (!collapse) {
             if (length(id >0)) d1name <- id else d1name <- NULL
             cluster <- d1name
             curve <- as.integer(X)
         }
         else {
             d1name <- unique(id)
             cluster <- match(id, d1name)
             curve <- (as.integer(X))[!duplicated(id)]
         }
```

```
            resid <- rsurvpart2(newY, X, casewt, istate, times, cluster,
                                type, object, method=method, collapse=collapse)

            if (type == "cumhaz") {
                ntemp <- colnames(object$cumhaz)
                if (length(dim(resid)) ==3)
                     dimnames(resid) <- list(id=d1name, times=timelab,
                                             cumhaz= ntemp)
                else dimnames(resid) <- list(id=d1name, cumhaz=ntemp)
            }
            else {
                ntemp <- object$states
                if (length(dim(resid)) ==3)
                    dimnames(resid) <- list(id=d1name, times=timelab,
                                            state= ntemp)
                else dimnames(resid) <- list(id=d1name, state= ntemp)
            }
        }
    }
    else stop("coxph survival curves not yet available")

    if (weighted && any(casewt !=1)) resid <- resid*casewt

    list(residuals= resid, curve= curve, id= id, idname=idname)
}
```

The first part of the work is retrieve the data set. This is done in multiple places in the survival code, all essentially the same. If I gave up (like lm) and forced the model frame to be saved this would be easier of course.

⟨*rsurvfit-data*⟩=

```
Call <- object$call

# remember the name of the id variable, if present.
#  but we don't try to parse it:  id= mydata$clinic becomes NULL
idname <- Call$id
if (is.name(idname)) idname <- as.character(idname)
else idname <- NULL
# I always need the model frame
if (coxsurv) {
    mf <- model.frame(object)
    if (is.null(object$y)) Y <- model.response(mf)
    else Y <- object$y
}
else {
    formula <- formula(object)
    # the chunk below is shared with survfit.formula
    na.action <- getOption("na.action")
    if (is.character(na.action))
        na.action <- get(na.action)  # this is a temporary hack
    ⟨survfit.formula-getdata⟩
    # end of shared code
}

xlev <- levels(X)

# Deal with ties
if (is.null(Call$timefix) || Call$timefix) newY <- aeqSurv(Y) else newY <- Y
```

This code has 3 primary sections: single state survival, multi-state survival, and post-Cox survival. A motivating idea in all of them is to avoid an $O(nd)$ calculation that involves the increment to each subject's leverage at each of the d event times. Since d often grows with n this can get very slow. This routine is designed for the case where the number of time points in the output matrix is modest, so we aim for $O(n)$ processes that repeat for each output time.

7.2 Simple survival

The Nelson-Aalen estimate of cumulative hazard is a simple sum

$$\begin{aligned} H(t) &= H(t-) + h(t) \\ \frac{\partial H(t)}{\partial w_i} &= \frac{\partial H(t-)}{\partial w_i} + [dN_i(t) - Y_i(t)h(t)]/r(t) \\ &= \sum_{d_j \le t} dN_i(d_j)/r(d_j) - Y_i(d_j)h(d_j)/r(d_j) \qquad (14) \end{aligned}$$

where H the cumulative hazard, h is the increment to the cumulative hazard, Y_i is 1 when a subject is at risk, and dN_i marks an event for the subject. Our basic strategy for the NA estimate is to use a two stage estimate. First, compute three vectors, each with one element per event time.

- term1 = $1/r(d_j)$ is the increment to the derivative for any observation with an event at event time d_j
- term2 = $-h(d_j)/r(d_j)$ is the increment for any observation that is at risk at time d_j
- term3 = cumulative sum of term2

For any given observation i whose follow-up interval is (s_i, t_i), their derivative at time z is the sum of

- term3(min(z, t_i)) - term3(min(z, s_i))
- term1(t_i) if $t_i \le z$ and observation i is an event

The Fleming-Harrington estimate of survival is

$$S(t) = e^{-H(t)}$$
$$\frac{\partial S(t)}{\partial w_i} = -S(t)\frac{\partial H(t)}{\partial w_i}$$

So has exactly the same computation, with a multiplication at the end.

⟨*residuals.survfit*⟩=
```
 rsurvpart1 <- function(Y, X, casewt, times,
          type, stype, ctype, fit) {

     ntime <- length(times)
     etime <- (fit$n.event >0)
     ny <- ncol(Y)
     event <- (Y[,ny] >0)
     status <- Y[,ny]

     #
     #  Create a list whose first element contains the location of
     #   the death times in curve 1, second element the death times for curve 2,
     #
     if (is.null(fit$strata)) {
         fitrow <- list(which(etime))
     }
     else {
         temp1 <- cumsum(fit$strata)
         temp2 <- c(1, temp1+1)
         fitrow <- lapply(1:length(fit$strata), function(i) {
             indx <- seq(temp2[i], temp1[i])
             indx[etime[indx]] # keep the death times
         })
     }
     ff <- unlist(fitrow)

     # for each time x, the index of the last death time which is <=x.
     #  0 if x is before the first death time in the fit object.
     #  The result is an index to the survival curve
     matchfun <- function(x, fit, index) {
         dtime <- fit$time[index]  # subset to this curve
         i2 <- findInterval(x, dtime, left.open=FALSE)
         c(0, index)[i2 +1]
     }

     # output matrix D will have one row per observation, one col for each
     #  reporting time. tindex and yindex have the same dimension as D.
```

```
    # tindex points to the last death time in fit which
    #  is <= the reporting time.  (If there is only 1 curve, each col of
    #  tindex will be a repeat of the same value.)
    tindex <- matrix(0L, nrow(Y), length(times))
    for (i in 1:length(fitrow)) {
        yrow <- which(as.integer(X) ==i)
        temp <- matchfun(times, fit, fitrow[[i]])
        tindex[yrow, ] <- rep(temp, each= length(yrow))
    }
    tindex[,] <- match(tindex, c(0,ff)) -1L  # the [,] preserves dimensions

    # repeat the indexing for Y onto fit$time.  Each row of yindex points
    #  to the last row of fit with death time <= Y[,ny]
    ny <- ncol(Y)
    yindex <- matrix(0L, nrow(Y), length(times))
    event <- (Y[,ny] >0)
    if (ny==3) startindex <- yindex
    for (i in 1:length(fitrow)) {
        yrow <- (as.integer(X) ==i)  # rows of Y for this curve
        temp <- matchfun(Y[yrow,ny-1], fit, fitrow[[i]])
        yindex[yrow,] <- rep(temp, ncol(yindex))
        if (ny==3) {
            temp <- matchfun(Y[yrow,1], fit, fitrow[[i]])
            startindex[yrow,] <- rep(temp, ncol(yindex))
        }
    }
    yindex[,] <- match(yindex, c(0,ff)) -1L
    if (ny==3) {
        startindex[,] <- match(startindex, c(0,ff)) -1L
        # no subtractions for report times before subject's entry
        startindex <- pmin(startindex, tindex)
    }

    # Now do the work
    if (type=="cumhaz" || stype==2) {  # result based on hazards
        if (ctype==1) {
            ⟨residpart1-nelson⟩
        } else {
            ⟨residpart1-fleming⟩
        }
    } else { # not hazard based
        ⟨residpart1-AJ⟩
    }
    D
}
```

The Nelson-Aalen is the simplest case. We don't have to worry about case weights of the data, since that has already been accounted for by the survfit function.

```
⟨residpart1-nelson⟩=
 death <- (yindex <= tindex & rep(event, ntime)) # an event occured at <= t

 term1 <- 1/fit$n.risk[ff]
 term2 <- lapply(fitrow, function(i) fit$n.event[i]/fit$n.risk[i]^2)
 term3 <- unlist(lapply(term2, cumsum))

 sum1 <- c(0, term1)[ifelse(death, 1+yindex, 1)]
 sum2 <- c(0, term3)[1 + pmin(yindex, tindex)]
 if (ny==3) sum3 <- c(0, term3)[1 + pmin(startindex, tindex)]

 if (ny==2) D <- matrix(sum1 -  sum2, ncol=ntime)
 else       D <- matrix(sum1 + sum3 - sum2, ncol=ntime)

 # survival is exp(-H) so the derivative is a simple transform of D
 if (type== "pstate") D <- -D* c(1,fit$surv[ff])[1+ tindex]
 else if (type == "auc") {
     ⟨auctrick⟩
 }
```

The sojourn time is the area under the survival curve. Let x_j be the widths of the rectangles under the curve from event time d_j to $\min(d_{j+1}, t)$, zero if $t \le d_j$, or $t - d_m$ if t is after the last event time.

$$\begin{aligned}
A(0,t) &= \sum_{j=1}^{m} x_j S(d_j) \\
\frac{\partial A(0,t)}{\partial w_i} &= \sum_{j=1}^{m} -x_j S(d_j) \frac{\partial H(d_j)}{\partial w_i} \\
&= \sum_{j=1}^{m} -x_j S(d_j) \sum_{k \le j} \frac{\partial h(d_k)}{\partial w_i} \\
&= \sum_{k=1}^{m} \frac{\partial h(d_k)}{\partial w_i} \left(\sum_{j \ge k} -x_j S(d_j) \right) \\
&= \sum_{k=1}^{m} -A(d_k, t) \frac{\partial h(d_k)}{\partial w_i}
\end{aligned}$$

For an observation at risk over the interval (a, b) we have exactly the same calculus as the cumulative hazard with respect to which $h(d_k)$ terms are counted for the observation, but now they are weighted sums. The weights are different for each output time, so we set them up as a matrix. We need the AUC at each event time d_k, and the AUC at the output times.

Matrix subscripts are a little used feature of R. If y is a matrix of values and x is a 2 colum matrix containing m (row, col) pairs, the result will be a vector of length m that plucks out the [x[1,1], x[1,2]] value of y, then the [x[2,1], x[2,2]] value of y, etc. They are rarely useful, but very handy in the few cases where they apply.

```
⟨auctrick⟩=
 auc1 <- lapply(fitrow, function(i) {
              if (length(i) <=1) 0
              else c(0, cumsum(diff(fit$time[i]) * (fit$surv[i])[-length(i)]))
                  })  # AUC at each event time
 auc2 <- lapply(fitrow, function(i) {
              if (length(i) <=1) 0
              else {
                  xx <- sort(unique(c(fit$time[i], times))) # all the times
                  yy <- (fit$surv[i])[findInterval(xx, fit$time[i])]
                  auc <- cumsum(c(diff(xx),0) * yy)
                  c(0, auc)[match(times, xx)]
                  }})  # AUC at the output times

 # Most often this function is called with a single curve, so make that case
 #  faster.  (Or I presume so: mapply and do.call may be more efficient than
 #  I think for lists of length 1).
 if (length(fitrow)==1) { # simple case, most common to ask for auc
     wtmat <- pmin(outer(auc1[[1]], -auc2[[1]], ’+’),0)
     term1 <- term1 * wtmat
     term2 <- unlist(term2) * wtmat
     term3 <- apply(term2, 2, cumsum)
 }
 else { #more than one curve, compute weighted cumsum per curve
     wtmat <- mapply(function(x, y) pmin(outer(x, -y, "+"), 0), auc1, auc2)
     term1 <- term1 * do.call(rbind, wtmat)
     temp <- mapply(function(x, y) apply(x*y, 2, cumsum), term2, wtmat)
     term3 <- do.call(rbind, temp)
 }

 sum1 <- sum2 <- matrix(0, nrow(yindex), ntime)
 if (ny ==3) sum3 <- sum1
 for (i in 1:ntime) {
     sum1[,i] <- c(0, term1[,i])[ifelse(death[,i], 1 + yindex[,i], 1)]
     sum2[,i] <- c(0, term3[,i])[1 + pmin(yindex[,i], tindex[,i])]
     if (ny==3) sum3[,i] <- c(0, term3[,i])[1 + pmin(startindex[,i], tindex[,i])]
 }
 # Perhaps a bit faster(?), but harder to read. And for AUC people usually only
 #  ask for one time point
 #sum1 <- rbind(0, term1)[cbind(c(ifelse(death, 1+yindex, 1)), c(col(yindex)))]
 #sum2 <- rbind(0, term3)[cbind(c(1 + pmin(yindex, tindex)), c(col(yindex)))]
```

```
#if (ny==3) sum3 <-
#             rbind(0, term3)[c(cbind(1 + pmin(startindex, tindex)),
#                               c(col(yindex)))]
if (ny==2) D <- matrix(sum1 -  sum2, ncol=ntime)
else       D <- matrix(sum1 + sum3 - sum2, ncol=ntime)
```

Fleming-Harrington For the Fleming-Harrington estimator the calculation at a tied time differs slightly. If there were 10 at risk and 3 tied events, the Nelson-Aalen has an increment of 3/10, while the FH has an increment of $(1/10 + 1/9 + 1/8)$. The underlying idea is that the true time values are continuous and we observe ties due to coarsening of the data. The derivative will have 3 terms as well. In this case the needed value cannot be pulled directly from the survfit object. Computationally, the number of distinct times at which a tie occurs is normally quite small and the for loop below will not be too expensive.

```
⟨residpart1-fleming⟩=
stop("residuals function still imcomplete, for FH estimate")
if (any(casewt != casewt[1])) {
    # Have to reconstruct the number of obs with an event, the curve only
    # contains the weighted sum
    nevent <- unlist(lapply(seq(along.with=levels(X)), function(i) {
        keep <- which(as.numeric(X) ==i)
        counts <- table(Y[keep, ny-1], status)
        as.vector(counts[, ncol(counts)])
        }))
} else nevent <- fit$n.event

n2 <- fit$n.risk
risk2 <- 1/fit$n.risk
ltemp <- risk2^2
for (i in which(nevent>1)) {  # assume not too many ties
    denom <- fit$n.risk[i] - fit$n.event[i]*(0:(nevent[i]-1))/nevent[i]
    risk2[i] <- mean(1/denom) # multiplier for the event
    ltemp[i] <- mean(1/denom^2)
    n2[i] <- mean(denom)
}

death <- (yindex <= tindex & rep(event, ntime))
term1 <- risk2[ff]
term2 <- lapply(fitrow, function(i) event[i]*ltemp[i])
term3 <- unlist(lapply(term2, cumsum))

sum1 <- c(0, term1)[ifelse(death, 1+yindex, 1)]
sum2 <- c(0, term3)[1 + pmin(yindex, tindex)]
if (ny==3) sum3 <- c(0, term3)[1 + pmin(startindex, tindex)]

if (ny==2) D <- matrix(sum1 -  sum2, ncol=ntime)
```

```
else       D <- matrix(sum1 + sum3 - sum2, ncol=ntime)

if (type=="pstate") D <- -D* c(0,fit$surv[ff])[1+ tindex]
else if (type=="auc") {
    ⟨auctrick⟩
}
```

Kaplan-Meier For the Kaplan-Meier (a special case of the Aalen-Johansen) the underlying algorithm is multiplicative, but we can turn it into an additive algoritm with a slight of hand.

$$
\begin{aligned}
S(t) &= \prod_{d_j \le t} (1 - h(d_j)) \\
&= \exp\left(\sum_{d_j \le t} \log(1 - h(d_j)) \right) \\
&= \exp\left(\sum_{d_j \le t} \log(r(d_j) - dN(d_j)) - \log(r(d_j)) \right) \\
\frac{\partial S(t)}{\partial w_i} &= S(t) \sum_{d_j \le t} \frac{Y_i(d_j) - dN_i(d_j)}{r(d_j) - dN(d_j)} - \frac{Y_i(d_j)}{r(d_j)}
\end{aligned}
$$

The addend for term2 is now $1/n(n-e)$ where e is the number of events, i.e., the same term as in the Greenwood variance, and term1 is $-1/n(n-e)$. The jumps in the KM curve are just a big larger than jumps in a FH estimate, so it makes sense that these are just a bit larger.

```
⟨residpart1-AJ⟩=
 death <- (yindex <= tindex & rep(event, ntime))
 # dtemp avoids 1/0.  (When this occurs the influence is 0, since
 #  the curve has dropped to zero; and this avoids Inf in term1 and term2).
 dtemp <- ifelse(fit$n.risk==fit$n.event, 0, 1/(fit$n.risk- fit$n.event))
 term1 <- dtemp[ff]
 term2 <- lapply(fitrow, function(i) dtemp[i]*fit$n.event[i]/fit$n.risk[i])
 term3 <- unlist(lapply(term2, cumsum))

 add1 <- c(0, term1)[ifelse(death, 1+yindex, 1)]
 add2 <- c(0, term3)[1 + pmin(yindex, tindex)]
 if (ny==3) add3 <- c(0, term3)[1 + pmin(startindex, tindex)]

 if (ny==2) D <- matrix(add1 -  add2, ncol=ntime)
 else       D <- matrix(add1 + add3 - add2, ncol=ntime)

 # survival is exp(-H) so the derivative is a simple transform of D
 if (type== "pstate") D <- -D* c(1,fit$surv[ff])[1+ tindex]
 else if (type == "auc") {
    ⟨auctrick⟩
}
```

7.3 Multi-state Aalen-Johansen estimate

For multi-state models a correction for ties of similar spirit to the Efron approximation in a Cox model (the ctype=2 argument for `survfit`) is difficult: the ‘right’ answer depends on the study. Thus the ctype argument is not present. Both stype 1 and 2 are feasible, but currently only `stype=1` is supported. This makes the code somewhat simpler, but this is more than offset by the multi-state nature. With multiple states we also need to account for influence on the starting state $p(0)$.

One thing that can make this code slow is data that has been divided into a very large number of intervals, giving a large number of observations for each cluster. We first deal with that by collapsing adjacent observations.

```
⟨residuals.survfit⟩=
rsurvpart2 <- function(Y, X, casewt, istate, times, cluster, type, fit,
                       method, collapse) {
    ny <- ncol(Y)
    ntime <- length(times)
    nstate <- length(fit$states)

    # ensure that Y, istate, and fit all use the same set of states
    states <- fit$states
    if (!identical(attr(Y, "states"), fit$states)) {
        map <- match(attr(Y, "states"), fit$states)
        Y[,ny] <- c(0, map)[1+ Y[,ny]]    # 0 = censored
        attr(Y, "states") <- fit$states
    }
    if (is.null(istate)) istate <- rep(1L, nrow(Y)) #everyone starts in s0
    else {
        if (is.character(istate)) istate <- factor(istate)
        if (is.factor(istate)) {
            if (!identical(levels(istate), fit$states)) {
                map <- match(levels(istate), fit$states)
                if (any(is.na(map))) stop ("invalid levels in istate")
                istate <- map[istate]
            }
        } # istate is numeric, we take what we get and hope it is right
    }

    # collapse redundant rows in Y, for efficiency
    #  a redundant row is a censored obs in the middle of a chain of times
    #  If the user wants individial obs, however, we would just have to
    #  expand it again
    if (ny==3 && collapse & any(duplicated(cluster))) {
```

```
    ord <- order(cluster, Y[,1])  # time within subject
    cfit <- .Call(Ccollapse, Y, X, istate, cluster, casewt, ord -1L)
    if (nrow(cfit) < .8*length(X))  {
        # shrinking the data by 20 percent is worth it
        temp <- Y[ord,]
        Y <- cbind(temp[cfit[,1], 1], temp[cfit[2], 2:3])
        X <- X[cfit[,1]]
        istate <- istate[cfit[1,]]
        cluster <- cluster[cfit[1,]]
    }
}

# Compute the initial leverage
inf0 <- NULL
if (is.null(fit$call$p0) && any(istate != istate[1])) {
    #p0 was not supplied by the user, and the intitial states vary
    inf0 <- matrix(0., nrow=nrow(Y), ncol=nstate)
    i0fun <- function(i, fit, inf0) {
        # reprise algorithm in survfitCI
        p0 <- fit$p0
        t0 <- fit$time[1]
        if (ny==2) at.zero <- which(as.numeric(X) ==i)
        else
            at.zero <- which(as.numeric(X) ==i &
                      (Y[,1] < t0 & Y[,2] >= t0))
        for (j in 1:nstate) {
            inf0[at.zero, j] <- (ifelse(istate[at.zero]==states[j], 1, 0) -
                                 p0[j])/sum(casewt[at.zero])
        }
        inf0
    }

    if (is.null(fit$strata)) inf0 <- i0fun(1, fit, inf0)
    else for (i in 1:length(levels(X)))
        inf0 <- i0fun(i, fit[i], inf0)  # each iteration fills in some rows
}

p0 <- fit$p0          # needed for method==1, type != cumhaz
fit <- survfit0(fit)  # package the initial state into the picture
start.time <- fit$time[1]

# This next block is identical to the one in rsurvpart1, more comments are
#  there
etime <- (rowSums(fit$n.event) >0)
event <- (Y[,ny] >0)
#
```

```
    #  Create a list whose first element contains the location of
    #   the death times in curve 1, second element for curve 2, etc.
    #
    if (is.null(fit$strata)) fitrow <- list(which(etime))
    else {
        temp1 <- cumsum(fit$strata)
        temp2 <- c(1, temp1+1)
        fitrow <- lapply(1:length(fit$strata), function(i) {
            indx <- seq(temp2[i], temp1[i])
            indx[etime[indx]] # keep the death times
        })
    }
    ff <- unlist(fitrow)

    # for each time x, the index of the last death time which is <=x.
    #  0 if x is before the first death time
    matchfun <- function(x, fit, index) {
        dtime <- fit$time[index]  # subset to this curve
        i2 <- findInterval(x, dtime, left.open=FALSE)
        c(0, index)[i2 +1]
    }


    if (type== "cumhaz") {
        ⟨residpart2CH⟩
    } else {
        ⟨residpart2AJ⟩
    }

    # since we may have done a partial collapse (removing redundant rows), the
    # parent routine can't collapse the data
    if (collapse & any(duplicated(cluster))) {
        if (length(dim(D)) ==2)
            D <- rowsum(D, cluster, reorder=FALSE)
        else { #rowsums has to be fooled
            dd <- dim(D)
            temp <- rowsum(matrix(D, nrow=dd[1]), cluster)
            D <- array(temp, dim=c(nrow(temp), dd[2:3]))
        }
    }
    D
}
```

Nelson-Aalen The multi-state Nelson-Aalen estimate of the cumulative hazard at time t is a vector with one element for each observed transition pair. If there were k states there are potentially $k(k-1)$ transition pairs, though normally only a small number will occur in a given fit. We ignore transitions from state j to state j. Let $r(t)$ be the weighted number at risk at time t, in each state. When some subject makes a $j:k$ transition, the $j:k$ transition will have an increment of $w_i/r_j(t)$. This is precisely the same increment as the ordinary Nelson estimate. The only change then is that we loop over the set of possible transitions, creating a large output object.

⟨*residpart2CH*⟩=
```
 # output matrix D will have one row per observation, one col for each
 #  reporting time. tindex and yindex have the same dimension as D.
 # tindex points to the last death time in fit which
 #  is <= the reporting time.  (If there is only 1 curve, each col of
 #  tindex will be a repeat of the same value.)
 tindex <- matrix(0L, nrow(Y), length(times))
 for (i in 1:length(fitrow)) {
     yrow <- which(as.integer(X) ==i)
     temp <- matchfun(times, fit, fitrow[[i]])
     tindex[yrow, ] <- rep(temp, each= length(yrow))
 }
 tindex[,] <- match(tindex, c(0,ff)) -1L  # the [,] preserves dimensions

 # repeat the indexing for Y onto fit$time.  Each row of yindex points
 #  to the last row of fit with death time <= Y[,ny]
 ny <- ncol(Y)
 yindex <- matrix(0L, nrow(Y), length(times))
 event <- (Y[,ny] >0)
 if (ny==3) startindex <- yindex
 for (i in 1:length(fitrow)) {
     yrow <- (as.integer(X) ==i)  # rows of Y for this curve
     temp <- matchfun(Y[yrow,ny-1], fit, fitrow[[i]])
     yindex[yrow,] <- rep(temp, ncol(yindex))
     if (ny==3) {
         temp <- matchfun(Y[yrow,1], fit, fitrow[[i]])
         startindex[yrow,] <- rep(temp, ncol(yindex))
     }
 }
 yindex[,] <- match(yindex, c(0,ff)) -1L
 if (ny==3) {
     startindex[,] <- match(startindex, c(0, ff)) -1L
     # no subtractions for report times before subject's entry
     startindex <- pmin(startindex, tindex)
 }

 dstate <- Y[,ncol(Y)]
 istate <- as.integer(istate)
 ntrans <- ncol(fit$cumhaz)  # the number of possible transitions
```

```
D <- array(0, dim=c(nrow(Y), ntime, ntrans))

scount <- table(istate[dstate!=0], dstate[dstate!=0]) # observed transitions
state1 <- row(scount)[scount>0]
state2 <- col(scount)[scount>0]
temp <- paste(rownames(scount)[state1],
              colnames(scount)[state2], sep=’.’)
if (!identical(temp, colnames(fit$cumhaz))) stop("setup error")

for (k in length(state1)) {
    e2 <- Y[,ny] == state2[k]
    add1 <- (yindex <= tindex & rep(e2, ntime))
    lsum <- unlist(lapply(fitrow, function(i)
             cumsum(fit$n.event[i,k]/fit$n.risk[i,k]^2)))

    term1 <- c(0, 1/fit$n.risk[ff,k])[ifelse(add1, 1+yindex, 1)]
    term2 <- c(0, lsum)[1+pmin(yindex, tindex)]
    if (ny==3) term3 <- c(0, lsum)[1 + startindex]

    if (ny==2) D[,,k] <- matrix(term1 -  term2, ncol=ntime)
    else       D[,,k] <- matrix(term1 + term3 - term2, ncol=ntime)
}
```

Aalen-Johansen The multi-state AJ estimate is more complex. Let $p(t)$ be the vector of probability in state at time t. Then

$$\begin{aligned}
p(t) &= p(t-)[I + A(t)] \\
\frac{\partial p(t)}{\partial w_i} &= \frac{\partial p(t-)}{\partial w_i}[I + A(t)] + p(t-)\frac{\partial A(t)}{\partial w_i} \\
&= U_i(t-)[I + A(t)] + p(t-)\frac{\partial A(t)}{\partial w_i} \qquad (15)
\end{aligned}$$

When we expand the left hand portion of (15) to include all observations it becomes simple matrix multiplication, not so with the right hand portion. Each individual subject i has a subject-specific nstate * nstate derivative matrix dA, which will be non-zero only for the state (row) j that the subject occupies at time $t-$. The jth row of $p(t-)dH$ is added to each subject's derivative.

The A matrix at time t has off diagonal elements and derivative

$$\begin{aligned}
A(t)_{jk} &= \frac{\sum_i w_i Y_{ij}(t) dNik(t)}{\sum_i w_i Y_{ij}(t)} \qquad (16) \\
&= \lambda_{jk}(t) \qquad (17) \\
\frac{\partial A(t)}{\partial w_i} &= \frac{dN_{ik}(t) - \lambda_{jk}(t)}{\sum_i w_i Y_{ij}(t)} \qquad (18)
\end{aligned}$$

This is the standard counting process notation: $Y_{ij}(t)$ is 1 if subject i is in state j and at risk at time $t-$, and $dN_{ik}(t)$ is a transition to state k at time t. Each observation at risk appears in at most 1 row of $A(t)$, since they can only be in one state. The diagonal element of A are set so that each row sums to 0. If there are no transitions out of state j at some time point, then that row of A is zero. Since the row sums are constant, the sum of the derivatives for each row must be zero.

If we evaluate equation directly there will be $O(nk^2)$ operations at each death time for the matrix product, and another $O(nk)$ to add in the new increment. For a large data set d is often of the same order as n, which makes this an expensive calculation. But, this is what the C-code version currently does, because I have code that actually works.

```
⟨residpart2AJ⟩=
 if (method==1) {
     # Compute the result using the direct method, in C code
     # the routine is called separately for each curve, data in sorted order
     #
     is1 <- as.integer(istate) -1L  # 0 based subscripts for C
     if (is.null(inf0)) inf0 <- matrix(0, nrow=nrow(Y), ncol=nstate)
     if (all(as.integer(X) ==1)) { # only one curve
         if (ny==2) asort1 <- 0L else asort1 <- order(Y[,1], Y[,2]) -1L
         asort2 <- order(Y[,ny-1]) -1L
         tfit <- .Call(Csurvfitresid, Y, asort1, asort2, is1,
                       casewt, p0, inf0, times, start.time,
                       type== "auc")

         if (ntime==1) {
             if (type=="auc") D <- tfit[[2]] else D <- tfit[[1]]
         }
         else {
             if (type=="auc") D <- array(tfit[[2]], dim=c(nrow(Y), nstate, ntime))
             else         D <- array(tfit[[1]], dim=c(nrow(Y), nstate, ntime))
         }
     }
     else { # one curve at a time
         ix <- as.numeric(X)  # 1, 2, etc
         if (ntime==1) D <- matrix(0, nrow(Y), nstate)
         else D <- array(0, dim=c(nrow(Y), nstate, ntime))
         for (curve in 1:max(ix)) {
             j <- which(ix==curve)
             ytemp <- Y[j,,drop=FALSE]
             if (ny==2) asort1 <- 0L
             else asort1 <- order(ytemp[,1], ytemp[,2]) -1L
             asort2 <- order(ytemp[,ny-1]) -1L

             # call with a subset of the data
             j <- which(ix== curve)
```

```
            tfit <- .Call(Csurvfitresid, ytemp, asort1, asort2, is1[j],
                          casewt[j], p0[curve,], inf0[j,], times,
                          start.time, type=="auc")
            if (ntime==1) {
                if (type=="auc") D[j,] <- tfit[[2]] else D[j,] <- tfit[[1]]
            } else {
                if (type=="auc") D[j,,] <- tfit[[2]] else D[j,,] <- tfit[[1]]
            }
        }
    }
    # the C code makes time the last dimension, we want it to be second
    if (ntime > 1) D <- aperm(D, c(1,3,2))
}
else {
    # method 2
    ⟨residpart2AJ2⟩
}
```

Can we speed this up? An alternate is to look at the direct expansion.

$$
\begin{aligned}
p(t) &= p(0) \prod_{d_j \le t} [I + A(d_j)] \\
\frac{\partial p(t)}{\partial w_i} &= \frac{\partial p(0)}{\partial w_i} \prod_{d_j \le t} [I + A(d_j)] \qquad (19) \\
&\quad + p(0) \sum_{d_j \le t} \left(\prod_{k<j} [I + A(d_k)] \frac{\partial A(d_j)}{\partial w_i} \prod_{j<k, d_k \le t} [I + A(d_k)] \right) \\
&= \frac{\partial p(0)}{w_i} \prod_{d_j \le t} [I + A(d_j)] + p(d_{j-1}) \frac{\partial A(d_j)}{\partial w_i} \prod_{j<k, d_k \le t} [I + A(d_k)] \qquad (20)
\end{aligned}
$$

We cannot insert an $(I + A(d_j))/(I + A(d_j))$ term and rearrange the last equation so as to factor out $p(t)$, as was done in the KM case, however, since matrix products do not commute. Instead think of accumulating the terms sequentially. Let $B^{(j)}(t)$ be the nstate by nstate matrix derivative matrix with row j of $\lambda_{jk}/n_j(t)$, and zero in all of the other rows, i.e., term 2 of equation (18) for someone in state j. (This is the part of the derivative that is common to all subjects at risk.) Let $B(t)$ be the sum of these matrices, i.e., all states filled. Now, here is the trick. The product $B^{(j)}(t)[I + A(t)]$ also is zero for all but the jth row, and is in fact equal to the jth row of $B(t)[I + A(t)]$. Further, $p(t-)B^{(j)}(t)[I + A(t)]$ is the jth row of $\text{diag}(p(t-))B(t)[I + A(t)]$.

The key computation is based on a matrix of matrices. Start with the following definitions. T_{jk} is the jth term in the expansion, at death time k. $T_{jk} = 0$ whenever $k = 0$ or $j > k$. Let

$D(x)$ be the diagonal matrix.

$$T_{01} = D(p'(0))[I + A(d_1)] \qquad T_{02} = T_{01}[I + A(d_2)] \qquad T_{03} = T_{02}[I + A(d_3)] \qquad \ldots \tag{21}$$
$$T_{11} = D(p(d_1))B(d_1) \qquad T_{12} = T_{11}[I + A(d_2)] \qquad T_{13} = T_{12}[I + A(d_3)] \qquad \ldots \tag{22}$$
$$T_{21} = 0 \qquad T_{22} = D(p(d_2))B(d_2) \qquad T_{23} = T_{22}[I + A(d_2)] \qquad \ldots \tag{23}$$
$$T_{31} = 0 \qquad T_{32} = 0 \qquad T_{33} = D(p(d_3))B(d_3) \qquad \ldots \tag{24}$$

(According to the latex guide the above should be nicely spaced, but I get equations that are touching. Why?)

If $p(0)$ is a fixed value specified by the user then $p'(0)$ =0. Otherwise $p(0)$ is the emprical distribution of the initial states, just before the first death time d_1. Let n_0 be the (weighted) count of subjects who are at risk at that time. The jth row of $p'(0)$ is defined as the deviative wrt w_i for a subject who starts in state j. If no one starts in state j that row of the matrix will be 0, otherwise it contains $(1 - p_j(0)$ in the jth element and $p_j(0)/n_0$ elsewhere.

Define the matrix $W_{jk} = \sum_{l=1}^{j} T_{lk}$, with $W_{j0} = 0$. Then for someone who enters at time s such that $d_a < s \le d_{a+1}$, is censored or has an event at time t such that $d_b \le t < d_{b+1}$, reporting at time r such that $d_c \le r < d_{c+1}$, the first portion of the contribution for an observation in state j will be the jth row of $-(W_{br} - W_{ar})$.

The second contribution is the effect of the dN term in the derivative. An observation that has a j:k transtion at time d_i will have an additional term of $c \prod_{k=i+1}^{r} [I + A(t_k)]$ where c is a vector with

$$\begin{aligned} c_j &= -1/n_j(d_i) \\ c_k &= 1/n_j(d_i) \\ c &= 0 \text{ otherwise} \end{aligned}$$

If there are multiple reporting times, it is currently simplest to do each one separately (at least for now), having computed and stored the sets of matrices $A(d_i)$ and $p(d_i)B(d_i)$ once at the start. If there are multiple strata in a curve, this is done separately per stratum.

```
⟨residpart2AJ2⟩=
 Yold <- Y
 utime  <- fit$time[fit$time <= max(times) & etime] # unique death times
 ndeath <- length(utime)    # number of unique event times
 delta <- diff(c(start.time, utime))

 # Expand Y
 if (ny==2) split <- .Call(Csurvsplit, rep(0., nrow(Y)), Y[,1], times)
 else split <- .Call(Csurvsplit, Y[,1], Y[,2], times)
 X <- X[split$row]
 casewt <- casewt[split$row]
 istate <- istate[split$row]
 Y <- cbind(split$start, split$end,
             ifelse(split$censor, 0, Y[split$row,ny]))
 ny <- 3
```

```
# Create a vector containing the index of each end time into the fit object
yindex <- ystart <- double(nrow(Y))
for (i in 1:length(fitrow)) {
    yrow <- (as.integer(X) ==i)  # rows of Y for this curve
    yindex[yrow] <- matchfun(Y[yrow, 2], fit, fitrow[[i]])
    ystart[yrow] <- matchfun(Y[yrow, 1], fit, fitrow[[i]])
}
# And one indexing the reporting times into fit
tindex <- matrix(0L, nrow=length(fitrow), ncol=ntime)
for (i in 1:length(fitrow)) {
    tindex[i,] <- matchfun(times, fit, fitrow[[i]])
}
yindex[,] <- match(yindex, c(0,ff)) -1L
tindex[,] <- match(tindex, c(0,ff)) -1L
ystart[,] <- pmin(match(ystart, c(0,ff)) -1L, tindex)

# Create the array of C matrices
cmat <- array(0, dim=c(nstate, nstate, ndeath)) # max(i2) = ndeath, by design
Hmat <- cmat

# We only care about observations that had a transition; any transitions
#  after the last reporting time are not relevant
transition <- (Y[,ny] !=0 & Y[,ny] != istate &
               Y[,ny-1] <= max(times)) # obs that had a transition
i2 <- match(yindex, sort(unique(yindex)))  # which C matrix this obs goes to
i2 <- i2[transition]
from <- as.numeric(istate[transition])  # from this state
to   <- Y[transition, ny]   # to this state
nrisk <- fit$n.risk[cbind(yindex[transition], from)]  # number at risk
wt <- casewt[transition]
for (i in seq(along.with =from)) {
    j <- c(from[i], to[i])
    haz <- wt[i]/nrisk[i]
    cmat[from[i], j, i2[i]] <- cmat[from[i], j, i2[i]] + c(-haz, haz)
}
for (i in 1:ndeath) Hmat[,,i] <- cmat[,,i] + diag(nstate)

# The transformation matrix H(t) at time t  is cmat[,,t] + I
# Create the set of W and V matrices.
#
dindex <- which(etime & fit$time <= max(times))
Wmat <- Vmat <- array(0, dim=c(nstate, nstate, ndeath))
for (i in ndeath:1) {
    j <- match(dindex[i], tindex, nomatch=0)
    if (j > 0) {
        # this death matches one of the reporting times
        Wmat[,,i] <- diag(nstate)
        Vmat[,,i] <- matrix(0, nstate, nstate)
    }
    else {
        Wmat[,,i] <- Hmat[,,i+1] %*% Wmat[,,i+1]
        Vmat[,,i] <- delta[i] +  Hmat[,,i+1] %*% Wmat[,,i+1]
    }
}
```

The above code has created the Wmat array for all reporting times and for all the curves (if more than one). Each of them reaches forward to the next reporting time. Now work forward in time.

⟨*residpart2AJ2*⟩=
```
iterm <- array(0, dim=c(nstate, nstate, ndeath)) # term in equation
itemp <- vtemp <- matrix(0, nstate, nstate)  # cumulative sum, temporary
isum  <- isum2 <- iterm  # cumulative sum
vsum  <- vsum2 <- vterm <- iterm
for (i in 1:ndeath) {
    j <- dindex[i]
    n0 <- ifelse(fit$n.risk[j,] ==0, 1, fit$n.risk[j,]) # avoid 0/0
    iterm[,,i] <- ((fit$pstate[j-1,]/n0) * cmat[,,i]) %*% Wmat[,,i]
    vterm[,,i] <- ((fit$pstate[j-1,]/n0) * cmat[,,i]) %*% Vmat[,,i]
    itemp <- itemp + iterm[,,i]
    vtemp <- vtemp + vterm[,,i]
    isum[,,i] <- itemp
    vsum[,,i] <- vtemp
    j <- match(dindex[i], tindex, nomatch=0)
    if (j>0) itemp <- vtemp <- matrix(0, nstate, nstate)  # reset
    isum2[,,i] <- itemp
    vsum2[,,i] <- vtemp
}

# We want to add isum[state,, entry time] - isum[state,, exit time] for
#  each subject, and for those with an a:b transition there will be an
#  additional vector with -1, 1 in the a and b position.
i1 <- match(ystart, sort(unique(yindex)), nomatch=0) # start at 0 gives 0
i2 <- match(yindex, sort(unique(yindex)))
D <- matrix(0., nrow(Y), nstate)
keep <- (Y[,2] <= max(times))  # any intervals after the last reporting time
                                # will have 0 influence
for (i in which(keep)) {
    if (Y[i,3] !=0 && istate[i] != Y[i,3]) {
        z <- fit$pstate[yindex[i]-1, istate[i]]/fit$n.risk[yindex[i], istate[i]]
        temp <- double(nstate)
        temp[istate[i]] = -z
        temp[Y[i,3]]   =   z
        temp <- temp %*% Wmat[,,i2[i]] - isum[istate[i],,i2[i]]
        if (i1[i] >0) temp <- temp + isum2[istate[i],, i1[i]]
        D[i,] <- temp
    }
    else {
        if (i1[i] >0) D[i,] = isum2[istate[i],,i1[i]] - isum[istate[i],, i2[i]]
        else  D[i,] =  -isum[istate[i],, i2[i]]
    }
}
```

By design, each row of Y, and hence each row of D, corresponds to a unique curve, and also to a unique period in the reporting intervals. (Any Y intervals after the last reporting time will have D=0 for the row.) If there are multiple reporting intervals, create an array with one n by nstate slice for each. If a row lies in the first interval, D currently contains its influence on that interval. It's influence on the second interval is the vector times $\prod H(d_k)$ where k is the set of event times $>$ the first reporting time and $\leq$ the second one.

⟨*residpart2AJ2*⟩=

```
Dsave <- D
if (!is.null(inf0)) {
    # add in the initial influence, to the first row of each obs
    #   (inf0 was created on unsplit data)
    j <- which(!duplicated(split$row))
    D[j,] <- D[j,] + (inf0%*% Hmat[,,1] %*% Wmat[,,1])
}
if (ntime > 1) {
    interval <- findInterval(yindex, tindex, left.open=TRUE)
    D2 <- array(0., dim=c(dim(D), ntime))
    D2[interval==0,,1] <- D[interval==0,]
    for (i in 1:(ntime-1)) {
        D2[interval==i,,i+1] = D[interval==i,]
        j <- tindex[i]
        D2[,,i+1] = D2[,,i+1] + D2[,,i] %*% (Hmat[,,j] %*% Wmat[,,j])
    }
    D <- D2
}

# undo any artificial split
if (any(duplicated(split$row))) {
    if (ntime==1) D <- rowsum(D, split$row)
    else {
        # rowsums has to be fooled
        temp <- rowsum(matrix(D, ncol=(nstate*ntime)), split$row)
        # then undo it
        D <- array(temp, dim=c(nrow(temp), nstate, ntime))
    }
}
```

8 Accelerated Failure Time models

The `surveg` function fits parametric failure time models. This includes accerated failure time models, the Weibull, log-normal, and log-logistic models. It also fits as well as censored linear regression; with left censoring this is referred to in economics *Tobit* regression.

8.1 Residuals

The residuals for a `survreg` model are one of several types

response residual [[y]] value on the scale of the original data

deviance an approximate deviance residual. A very bad idea statistically, retained for the sake of backwards compatability.

dfbeta a matrix with one row per observation and one column per parameter showing the approximate influence of each observation on the final parameter value

dfbetas the dfbeta residuals scaled by the standard error of each coefficient

working residuals on the scale of the linear predictor

ldcase likelihood displacement wrt case weights

ldresp likelihood displacement wrt response changes

ldshape likelihood displacement wrt changes in shape

matrix matrix of derivatives of the log-likelihood wrt paramters

The other parameters are

rsigma whether the scale parameters should be included in the result for dfbeta results. I can think of no reason why one would not want them — unless of course the scale was fixed by the user, in which case there is no parameter.

collapse optional vector of subject identifiers. This is for the case where a subject has multiple observations in a data set, and one wants to have residuals per subject rather than residuals per observation.

weighted whether the residuals should be multiplied by the case weights. The sum of weighted residuals will be zero.

The routine starts with standard stuff, checking arguments for validity and etc. The two cases of response or working residuals require a lot less computation. and are the most common calls, so they are taken care of first.

⟨*residuals.survreg*⟩=

```
#
#  Residuals for survreg objects
residuals.survreg <- function(object, type=c('response', 'deviance',
                      'dfbeta', 'dfbetas', 'working', 'ldcase',
                      'ldresp', 'ldshape', 'matrix'),
                      rsigma =TRUE, collapse=FALSE, weighted=FALSE, ...) {
    type <-match.arg(type)
    n <- length(object$linear.predictors)
    Terms <- object$terms
    if(!inherits(Terms, "terms"))
            stop("invalid terms component of  object")

    # If the variance wasn't estimated then it has no error
    if (nrow(object$var) == length(object$coefficients)) rsigma <- FALSE

    # If there was a cluster directive in the model statment then remove
    #  it.  It does not correspond to a coefficient, and would just confuse
    #  things later in the code.
    cluster <- untangle.specials(Terms,"cluster")$terms
    if (length(cluster) >0 )
        Terms <- Terms[-cluster]

    strata <- attr(Terms, 'specials')$strata
    intercept <- attr(Terms, "intercept")
    response  <- attr(Terms, "response")
    weights <- object$weights
    if (is.null(weights)) weighted <- FALSE

    ⟨rsr-data⟩
    ⟨rsr-dist⟩
    ⟨rsr-resid⟩
    ⟨rsr-finish⟩
    }
```

First retrieve the distribution, which is used multiple times. The common case is a character string pointing to some element of `survreg.distributions`, but the other is a user supplied list of the form contained there. Some distributions are defined as the transform of another in which case we need to set `itrans` and `dtrans` and follow the link, otherwise the transformation and its inverse are the identity.

⟨*rsr-dist*⟩=

```
if (is.character(object$dist))
            dd <- survreg.distributions[[object$dist]]
else dd <- object$dist
ytype <- attr(y, "type")
if (is.null(dd$itrans)) {
    itrans <- dtrans <-function(x)x
    # reprise the work done in survreg to create a transformed y
    if (ytype==’left’) y[,2] <- 2- y[,2]
    else if (type==’interval’ && all(y[,3]<3)) y <- y[,c(1,3)]
}
else {
    itrans <- dd$itrans
    dtrans <- dd$dtrans

    # reprise the work done in survreg to create a transformed y
    tranfun <- dd$trans
    exactsurv <- y[,ncol(y)] ==1
    if (any(exactsurv)) logcorrect <-sum(log(dd$dtrans(y[exactsurv,1])))

    if (ytype==’interval’) {
        if (any(y[,3]==3))
            y <- cbind(tranfun(y[,1:2]), y[,3])
        else y <- cbind(tranfun(y[,1]), y[,3])
    }
    else if (ytype==’left’)
        y <- cbind(tranfun(y[,1]), 2-y[,2])
    else     y <- cbind(tranfun(y[,1]), y[,2])
}

if (!is.null(dd$dist))  dd <- survreg.distributions[[dd$dist]]
deviance <- dd$deviance
dens <- dd$density
```

The next task is to decide what data we need. The response is always needed, but is normally saved as a part of the model. If it is a transformed distribution such as the Weibull (a transform of the extreme value) the saved object `y` is the transformed data, so we need to replicate that part of the survreg() code. (Why did I even allow for y=F in survreg? Because I was mimicing the lm function — oh the long, long consequences of a design decision.)

The covariate matrix `x` will be needed for all but response, deviance, and working residuals. If the model included a strata() term then there will be multiple scales, and the strata variable needs to be recovered. The variable `sigma` is set to a scalar if there are no strata, but otherwise to a vector with `n` elements containing the appropriate scale for each subject.

The leverage type residuals all need the second derivative matrix. If there was a `cluster` statement in the model this will be found in `naive.var`, otherwise in the `var` component.

⟨*rsr-data*⟩=
```
if (is.null(object$naive.var)) vv <- object$var
else                           vv <- object$naive.var

need.x <- is.na(match(type, c(’response’, ’deviance’, ’working’)))
if (is.null(object$y) || !is.null(strata) || (need.x & is.null(object[[’x’]])))
    mf <- stats::model.frame(object)
```

```
 if (is.null(object$y)) y <- model.response(mf)
 else  y <- object$y

 if (!is.null(strata)) {
     temp <- untangle.specials(Terms, 'strata', 1)
     Terms2 <- Terms[-temp$terms]
     if (length(temp$vars)==1) strata.keep <- mf[[temp$vars]]
     else strata.keep <- strata(mf[,temp$vars], shortlabel=TRUE)
     strata <- as.numeric(strata.keep)
     nstrata <- max(strata)
     sigma <- object$scale[strata]
     }
 else {
     Terms2 <- Terms
     nstrata <- 1
     sigma <- object$scale
     }

 if (need.x) {
    x <- object[['x']]  #don't grab xlevels component
    if (is.null(x))
         x <- model.matrix(Terms2, mf, contrasts.arg=object$contrasts)
     }
```

The most common residual is type response, which requires almost no more work, for the others we need to create the matrix of derivatives before proceeding. We use the `center` component from the deviance function for the distribution, which returns the data point `y` itself for an exact, left, or right censored observation, and an appropriate midpoint for interval censored ones.

```
⟨rsr-resid⟩=
 if (type=='response') {
     yhat0 <- deviance(y, sigma, object$parms)
     rr <-  itrans(yhat0$center) - itrans(object$linear.predictor)
     }
 else {
     ⟨rtr-deriv⟩
     ⟨rtr-resid2⟩
     }
```

The matrix of derviatives is used in all of the other cases. The starting point is the `density` function of the distribtion which return a matrix with columns of $F(x)$, $1-F(x)$, $f(x)$, $f'(x)/f(x)$ and $f''(x)/f(x)$. The matrix type residual contains columns for each of

$$L_i \quad \frac{\partial L_i}{\partial \eta_i} \quad \frac{\partial^2 L_i}{\partial \eta_i^2} \quad \frac{\partial L_i}{\partial \log(\sigma)} \quad \frac{\partial L_i}{\partial \log(\sigma)^2} \quad \frac{\partial^2 L_i}{\partial \eta \partial \log(\sigma)}$$

where L_i is the contribution to the log-likelihood from each individual. Note that if there are multiple scales, i.e. a strata() term in the model, then terms 3–6 are the derivatives for that subject with respect to their *particular* scale factor; derivatives with respect to all the other scales are zero for that subject.

The log-likelihood can be written as

$$\begin{aligned}
L &= \sum_{exact} [\log(f(z_i)) - \log(\sigma_i)] + \sum_{censored} \log\left(\int_{z_i^l}^{z_i^u} f(u)du\right) \\
&\equiv \sum_{exact} [g_1(z_i) - \log(\sigma_i)] + \sum_{censored} \log(g_2(z_i^l, z_i^u)) \\
z_i &= (y_i - \eta_i)/\sigma_i
\end{aligned}$$

For the interval censored observations we have a z defined at both the lower and upper endpoints. The linear predictor is $\eta = X\beta$.

The derivatives are shown below. Note that $f(-\infty) = f(\infty) = F(-\infty) = 0$, $F(\infty) = 1$, $z^u = \infty$ for a right censored observation and $z^l = -\infty$ for a left censored one.

$$\begin{aligned}
\frac{\partial g_1}{\partial \eta} &= -\frac{1}{\sigma}\left[\frac{f'(z)}{f(z)}\right] \\
\frac{\partial g_2}{\partial \eta} &= -\frac{1}{\sigma}\left[\frac{f(z^u) - f(z^l)}{F(z^u) - F(z^l)}\right] \\
\frac{\partial^2 g_1}{\partial \eta^2} &= \frac{1}{\sigma^2}\left[\frac{f''(z)}{f(z)}\right] - (\partial g_1/\partial\eta)^2 \\
\frac{\partial^2 g_2}{\partial \eta^2} &= \frac{1}{\sigma^2}\left[\frac{f'(z^u) - f'(z^l)}{F(z^u) - F(z^l)}\right] - (\partial g_2/\partial\eta)^2 \\
\frac{\partial g_1}{\partial \log\sigma} & \qquad\qquad -\left[\frac{zf'(z)}{f(z)}\right] \\
\frac{\partial g_2}{\partial \log\sigma} &= -\left[\frac{z^u f(z^u) - z^l f(z^l)}{F(z^u) - F(z^l)}\right] \\
\frac{\partial^2 g_1}{\partial (\log\sigma)^2} &= \qquad\qquad \left[\frac{z^2 f''(z) + zf'(z)}{f(z)}\right] - (\partial g_1/\partial \log\sigma)^2 \\
\frac{\partial^2 g_2}{\partial (\log\sigma)^2} &= \left[\frac{(z^u)^2 f'(z^u) - (z^l)^2 f'(z_l)}{F(z^u) - F(z^l)}\right] - \partial g_1/\partial\log\sigma(1 + \partial g_1/\partial\log\sigma) \\
\frac{\partial^2 g_1}{\partial\eta\partial\log\sigma} &= \frac{zf''(z)}{\sigma f(z)} - \partial g_1/\partial\eta(1 + \partial g_1/\partial\log\sigma) \\
\frac{\partial^2 g_2}{\partial\eta\partial\log\sigma} &= \frac{z^u f'(z^u) - z^l f'(z^l)}{\sigma[F(z^u) - F(z^l)]} - \partial g_2/\partial\eta(1 + \partial g_2/\partial\log\sigma)
\end{aligned}$$

In the code `z` is the relevant point for exact, left, or right censored data, and `z2` the upper endpoint for an interval censored one. The variable `tdenom` contains the denominator for each subject (which is the same for all derivatives for that subject). For an interval censored observation we try to avoid numeric cancellation by using the appropriate tail of the distribution. For instance with $(z^l, z^u) = (12, 15)$ the value of $F(x)$ will be very near 1 and it is better to subtract two upper tail values $(1 - F)$ than two lower tail ones F.

⟨*rtr-deriv*⟩=
```
 status <- y[,ncol(y)]
 eta <- object$linear.predictors
 z <- (y[,1] - eta)/sigma
 dmat <- dens(z, object$parms)
 dtemp<- dmat[,3] * dmat[,4]    #f'
 if (any(status==3)) {
     z2 <- (y[,2] - eta)/sigma
     dmat2 <- dens(z2, object$parms)
     }
 else {
     dmat2 <- dmat   #dummy values
     z2 <- 0
     }

 tdenom <- ((status==0) * dmat[,2]) +  #right censored
           ((status==1) * 1 )       +  #exact
           ((status==2) * dmat[,1]) +  #left
           ((status==3) * ifelse(z>0, dmat[,2]-dmat2[,2],
                                      dmat2[,1] - dmat[,1])) #interval
 g <- log(ifelse(status==1, dmat[,3]/sigma, tdenom))  #loglik
 tdenom <- 1/tdenom
 dg <- -(tdenom/sigma) *(((status==0) * (0-dmat[,3])) +    #dg/ eta
                         ((status==1) * dmat[,4]) +
                         ((status==2) * dmat[,3]) +
                         ((status==3) * (dmat2[,3]- dmat[,3])))

 ddg <- (tdenom/sigma^2) *(((status==0) * (0- dtemp)) +  #ddg/eta^2
                           ((status==1) * dmat[,5]) +
                           ((status==2) * dtemp) +
                           ((status==3) * (dmat2[,3]*dmat2[,4] - dtemp)))

 ds  <- ifelse(status<3, dg * sigma * z,
                         tdenom*(z2*dmat2[,3] - z*dmat[,3]))
 dds <- ifelse(status<3, ddg* (sigma*z)^2,
                         tdenom*(z2*z2*dmat2[,3]*dmat2[,4] -
                                 z * z*dmat[,3] * dmat[,4]))
 dsg <- ifelse(status<3, ddg* sigma*z,
               tdenom *(z2*dmat2[,3]*dmat2[,4] - z*dtemp))
 deriv <- cbind(g, dg, ddg=ddg- dg^2,
                ds = ifelse(status==1, ds-1, ds),
                dds=dds - ds*(1+ds),
                dsg=dsg - dg*(1+ds))
```

Now, we can calcultate the actual residuals case by case. For the dfbetas there will be one column per coefficient, so if there are strata column 4 of the deriv matrix needs to be *un*collapsed into a matrix with nstrata columns. The same manipulation is needed for the ld residuals.

⟨*rtr-resid2*⟩=
```
 if (type=='deviance') {
     yhat0 <- deviance(y, sigma, object$parms)
     rr <- (-1)*deriv[,2]/deriv[,3]  #working residuals
     rr <- sign(rr)* sqrt(2*(yhat0$loglik - deriv[,1]))
     }

 else if (type=='working') rr <- (-1)*deriv[,2]/deriv[,3]

 else if (type=='dfbeta' || type== 'dfbetas' || type=='ldcase') {
     score <- deriv[,2] * x  # score residuals
     if (rsigma) {
         if (nstrata > 1) {
             d4 <- matrix(0., nrow=n, ncol=nstrata)
             d4[cbind(1:n, strata)] <- deriv[,4]
             score <- cbind(score, d4)
             }
         else score <- cbind(score, deriv[,4])
         }
     rr <- score %*% vv
     # cause column names to be retained
     # old: if (type=='dfbetas') rr[] <- rr %*% diag(1/sqrt(diag(vv)))
     if (type=='dfbetas') rr <- rr * rep(1/sqrt(diag(vv)), each=nrow(rr))
     if (type=='ldcase')  rr<- rowSums(rr*score)
     }

 else if (type=='ldresp') {
     rscore <-  deriv[,3] *  (x * sigma)
     if (rsigma) {
         if (nstrata >1) {
             d6 <- matrix(0., nrow=n, ncol=nstrata)
             d6[cbind(1:n, strata)] <- deriv[,6]*sigma
             rscore <- cbind(rscore, d6)
             }
         else rscore <- cbind(rscore, deriv[,6] * sigma)
         }
     temp <-  rscore %*% vv
     rr <- rowSums(rscore * temp)
     }

 else if (type=='ldshape') {
     sscore <- deriv[,6] *x
```

```
    if (rsigma) {
        if (nstrata >1) {
            d5 <- matrix(0., nrow=n, ncol=nstrata)
            d5[cbind(1:n, strata)] <- deriv[,5]
            sscore <- cbind(sscore, d5)
            }
        else sscore <- cbind(sscore, deriv[,5])
        }
    temp <- sscore %*% vv
    rr <- rowSums(sscore * temp)
    }

else {  #type = matrix
    rr <- deriv
    }
```

Finally the two optional steps of adding case weights and collapsing over subject id.

⟨*rsr-finish*⟩=

```
#case weights
if (weighted) rr <- rr * weights

#Expand out the missing values in the result
if (!is.null(object$na.action)) {
    rr <- naresid(object$na.action, rr)
    if (is.matrix(rr)) n <- nrow(rr)
    else               n <- length(rr)
    }

# Collapse if desired
if (!missing(collapse)) {
    if (length(collapse) !=n) stop("Wrong length for 'collapse'")
    rr <- drop(rowsum(rr, collapse))
    }

rr
```

9 Survival curves

The survfit function was set up as a method so that we could apply the function to both formulas (to compute the Kaplan-Meier) and to coxph objects. The downside to this is that the manual pages get a little odd, but from a programming perspective it was a good idea. At one time, long long ago, we allowed the function to be called with "Surv(time, status)" as the formula, i.e., without a tilde. That was a bad idea, now abandoned.

A note on times: one of the things that drove me nuts was the problem of "tied but not quite tied" times. As an example consider two values of 24173 = 23805 + 368. These are values from an actual study with times in days. However, the user chose to use age in years, and saved those values out in a CSV file, resulting in values for the above of 66.18206708000000 and 66.18206708000001. The R phrase `unique(x)` sees these two values as distinct but `table(x)` and `tapply` see it as a single value since they first apply `factor` to the values, and that in turn uses `as.character`. A transition through CSV is not necessary to create the problem:

```
⟨test⟩=
 tfun <- function(start, gap) {
     as.numeric(start)/365.25 - as.numeric(start + gap)/365.25
 }

 test <- logical(200)
 for (i in 1:200) {
     test[i] <- tfun(as.Date("2010/01/01"), 29) ==
                tfun(as.Date("2010/01/01") + i, 29)
 }
 table(test)
```

The number of FALSE entries in the table depends on machine, compiler, and a host of other issues. There is discussion of this general issue in the R FAQ: "why doesn't R think these numbers are equal". The Kaplan-Meier and Cox model both pay careful attention to ties, and so both now use the `aeqSurv` routine to first preprocess the time data. It uses the same rules as `all.equal` to adjudicate ties and near ties.

```
⟨survfit⟩=
 survfit <- function(formula, ...) {
     UseMethod("survfit")
 }

 ⟨survfit-formula⟩
 ⟨survfit-subscript⟩
 ⟨survfit-Surv⟩
```

The result of a survival curve will have a `surv` or `pstate` component that is a vector or a matrix, and an optional strata component. From a user's point of view this is an object with [strata, newdata, state] as dimensions, where only 1, 2 or all three of these may appear. The first is always present, and is essentially the number of distinct curves created by the right-hand side of the equation (or by the strata in a coxph model). The newdata portion appears for survival curves from a Cox model, when curves for multiple covariate patterns were requested; the state portion only from a multi-state model; or both for a multi-state Cox model. The `surv` component contains the time points for the first stratum, the second, third, etc stacked one above the other. As with R matrices, if only 1 subscript is given for an array or matrix of curves, we treat the collection of curves as a vector of curves. We need to make sure that the new object has the same order of elements as the old – users count on this.

```
⟨survfit-subscript⟩=
 dim.survfit <- function(x) {
    d1name <- "strata"
    d2name <- "data"
    d3name <- "states"
    if (is.null(x$strata))  {d1 <- d1name <- NULL} else d1 <- length(x$strata)
    if (is.null(x$newdata)) {d2 <- d2name <- NULL} else d2 <- nrow(x$newdata)
    if (is.null(x$states))  {d3 <- d3name <- NULL} else d3 <- length(x$states)

    if (inherits(x, "survfitcox") && is.null(d2) && is.null(d3) &&
        is.matrix(x$surv)) {
        # older style survfit.coxph object, before I added newdata to the output
        d2name <- "data"
        d2 <- ncol(x$surv)
    }

    dd <- c(d1, d2, d3)
    names(dd) <- c(d1name, d2name, d3name)
    dd
}

# there is a separate function for survfitms objects
"[.survfit" <- function(x, ... , drop=TRUE) {
    nmatch <- function(indx, target) {
        # This function lets R worry about character, negative, or
        #  logical subscripts.
        #  It always returns a set of positive integer indices
        temp <- 1:length(target)
        names(temp) <- target
        temp[indx]
    }

    if (!inherits(x, "survfit")) stop("[.survfit called on non-survfit object")
    ndots <- ...length()      # the simplest, but not avail in R 3.4
    # ndots <- length(list(...))# fails if any are missing, e.g. fit[,2]
    # ndots <- if (missing(drop)) nargs()-1 else nargs()-2  # a workaround

    dd <- dim(x)
    # for dd=NULL, an object with only one curve, x[1] is always legal
    if (is.null(dd)) dd <- c(strata=1L) # survfit object with only one curve
    dtype <- match(names(dd), c("strata", "data", "states"))

    if (ndots >0 && !missing(..1)) i <- ..1 else i <- NULL
    if (ndots> 1 && !missing(..2)) j <- ..2 else j <- NULL

    if (ndots > length(dd))
        stop("incorrect number of dimensions")
    if (length(dtype) > 2) stop("invalid survfit object")  # should never happen
```

```
 if (is.null(i) && is.null(j)) {
     # called with no subscripts given -- return x untouched
     return(x)
 }

 # Code below is easier if "i" is always the strata
 if (dtype[1] !=1) {
     dtype <- c(1, dtype)
     j <- i; i <- NULL
     dd <- c(1, dd)
     ndots <- ndots +1
 }

# We need to make a new one
 newx <- vector("list", length(x))
 names(newx) <- names(x)
 for (k in c("logse", "version", "conf.int", "conf.type", "type", "call"))
     if (!is.null(x[[k]])) newx[[k]] <- x[[k]]
 class(newx) <- class(x)

 if (ndots== 1 && length(dd)==2) {
     # one subscript given for a two dimensional object
     # If one of the dimensions is 1, it is easier for me to fill in i and j
     if (dd[1]==1) {j <- i; i<- 1}
     else if (dd[2]==1) j <- 1
     else {
         #  the user has a mix of rows/cols
         index <- 1:prod(dd)
         itemp <- matrix(index, nrow=dd[1])
         keep <- itemp[i]   # illegal subscripts will generate an error
         if (length(keep) == length(index) && all(keep==index)) return(x)

         ii <- row(itemp)[keep]
         jj <- col(itemp)[keep]
         # at this point we have a matrix subscript of (ii, jj)
         # expand into a long pair of rows and cols
         temp <- split(seq(along.with=x$time),
                       rep(1:length(x$strata), x$strata))
         indx1 <- unlist(temp[ii])   # rows of the surv object
         indx2 <- rep(jj, x$strata[ii])

         # return with each curve as a separate strata
         newx$n <- x$n[ii]
         for (k in c("time", "n.risk", "n.event", "n.censor", "n.enter"))
             if (!is.null(x[[k]])) newx[[k]] <- (x[[k]])[indx1]
         k <- cbind(indx1, indx2)
```

```
        for (j in c("surv", "std.err", "upper", "lower", "cumhaz",
                    "std.chaz", "influence.surv", "influence.chaz"))
            if (!is.null(x[[j]])) newx[[j]] <- (x[[j]])[k]
        temp <- x$strata[ii]
        names(temp) <- 1:length(ii)
        newx$strata <- temp
        return(newx)
    }
}

# irow will be the rows that need to be taken
#  j the columns (of present)
if (is.null(x$strata)) {
       if (is.null(i) || all(i==1)) irow <- seq(along.with=x$time)
       else stop("subscript out of bounds")
       newx$n <- x$n
}
else {
    if (is.null(i)) indx <- seq(along.with= x$strata)
    else indx <- nmatch(i, names(x$strata)) #strata to keep
    if (any(is.na(indx)))
        stop(paste("strata",
                   paste(i[is.na(indx)], collapse=' '),
                   'not matched'))
    # Now, indx may not be in order: some can use curve[3:2] to reorder
    #  The list/unlist construct will reorder the data
    temp <- split(seq(along.with =x$time),
                  rep(1:length(x$strata), x$strata))
    irow <- unlist(temp[indx])

    if (length(indx) <=1 && drop) newx$strata <- NULL
    else               newx$strata  <- x$strata[i]

    newx$n <- x$n[indx]
    if (length(indx) ==1 & drop) x$strata <- NULL
    else    newx$strata <- x$strata[indx]
}

if (length(dd)==1) {  # no j dimension
    for (k in c("time", "n.risk", "n.event", "n.censor", "n.enter",
           "surv", "std.err", "cumhaz", "std.chaz", "upper", "lower",
           "influence.surv", "influence.chaz"))
        if (!is.null(x[[k]])) newx[[k]] <- (x[[k]])[irow]
}

else { # 2 dimensional object
```

```
        if (is.null(j)) j <- seq.int(ncol(x$surv))
        # If the curve has been selected by strata and keep has only
        #  one row, we don't want to lose the second subscript too
        if (length(irow)==1)  drop <- FALSE

        for (k in c("time", "n.risk", "n.event", "n.censor", "n.enter"))
                 if (!is.null(x[[k]])) newx[[k]] <- (x[[k]])[irow]
        for (k in c("surv", "std.err", "cumhaz", "std.chaz", "upper", "lower",
               "influence.surv", "influence.chaz"))
            if (!is.null(x[[k]])) newx[[k]] <- (x[[k]])[irow, j, drop=drop]
    }
    newx
}
```

9.1 Kaplan-Meier

The most common use of the survfit function is with a formula as the first argument, and the most common outcome of such a call is a Kaplan-Meier curve.

The id argument is from an older version of the competing risks code; most people will use `cluster(id)` in the formula instead. The istate argument only applies to competing risks, but don't print an error message if it is accidentally there.

```
⟨survfit-formula⟩=
survfit.formula <- function(formula, data, weights, subset,
                            na.action, stype=1, ctype=1,
                            id, cluster, robust, istate,
                            timefix=TRUE, etype, error, ...) {

    Call <- match.call()
    Call[[1]] <- as.name('survfit')  #make nicer printout for the user
    ⟨survfit.formula-getdata⟩

    # Deal with the near-ties problem
    if (!is.logical(timefix) || length(timefix) > 1)
        stop("invalid value for timefix option")
    if (timefix) newY <- aeqSurv(Y) else newY <- Y

    if (missing(robust)) robust <- NULL
    # Call the appropriate helper function
    if (attr(Y, 'type') == 'left' || attr(Y, 'type') == 'interval')
        temp <-  survfitTurnbull(X, newY, casewt, cluster= cluster,
                                 robust= robust, ...)
    else if (attr(Y, 'type') == "right" || attr(Y, 'type')== "counting")
        temp <- survfitKM(X, newY, casewt, stype=stype, ctype=ctype, id=id,
                          cluster=cluster, robust=robust, ...)
    else if (attr(Y, 'type') == "mright" || attr(Y, "type")== "mcounting")
```

```
        temp <- survfitCI(X, newY, weights=casewt, stype=stype, ctype=ctype,
                          id=id, cluster=cluster, robust=robust,
                          istate=istate, ...)
    else {
        # This should never happen
        stop("unrecognized survival type")
    }

    # If a stratum had no one beyond start.time, the length 0 gives downstream
    #  failure, e.g., there is no sensible printout for summary(fit, time= 100)
    #  for such a curve
    temp$strata <- temp$strata[temp$strata >0]
    if (is.null(temp$states)) class(temp) <- 'survfit'
    else class(temp) <- c("survfitms", "survfit")

    if (!is.null(attr(mf, 'na.action')))
            temp$na.action <- attr(mf, 'na.action')

    temp$call <- Call
    temp
    }
```

This chunk of code is shared with resid.survfit

⟨*survfit.formula-getdata*⟩=

```
# create a copy of the call that has only the arguments we want,
#  and use it to call model.frame()
indx <- match(c('formula', 'data', 'weights', 'subset','na.action',
                'istate', 'id', 'cluster', "etype"), names(Call), nomatch=0)
#It's very hard to get the next error message other than malice
#  eg survfit(wt=Surv(time, status) ~1)
if (indx[1]==0) stop("a formula argument is required")
temp <- Call[c(1, indx)]
temp[[1L]] <- quote(stats::model.frame)
mf <- eval.parent(temp)

Terms <- terms(formula, c("strata", "cluster"))
ord <- attr(Terms, 'order')
if (length(ord) & any(ord !=1))
        stop("Interaction terms are not valid for this function")

n <- nrow(mf)
Y <- model.response(mf)
if (inherits(Y, "Surv2")) {
    # this is Surv2 style data
    # if there are any obs removed due to missing, remake the model frame
    if (length(attr(mf, "na.action"))) {
        temp$na.action <- na.pass
        mf <- eval.parent(temp)
    }
    if (!is.null(attr(Terms, "specials")$cluster))
        stop("cluster() cannot appear in the model statement")
    new <- surv2data(mf)
    mf <- new$mf
    istate <- new$istate
    id <- new$id
    Y <- new$y
    if (anyNA(mf[-1])) { #ignore the response variable still found there
        if (missing(na.action)) temp <- get(getOption("na.action"))(mf[-1])
        else temp <- na.action(mf[-1])
        omit <- attr(temp, "na.action")
        mf <- mf[-omit,]
        Y <- Y[-omit]
        id <- id[-omit]
        istate <- istate[-omit]
    }
    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("data set has no non-missing observations")

casewt <- model.extract(mf, "weights")
if (is.null(casewt)) casewt <- rep(1.0, n)
else {
    if (!is.numeric(casewt)) stop("weights must be numeric")
    if (any(!is.finite(casewt))) stop("weights must be finite")
    if (any(casewt <0)) stop("weights must be non-negative")
    casewt <- as.numeric(casewt)  # transform integer to numeric
}

if (!is.null(attr(Terms, ’offset’))) warning("Offset term ignored")

cluster <- model.extract(mf, "cluster")
temp <- untangle.specials(Terms, "cluster")
if (length(temp$vars)>0) {
    if (length(cluster) >0) stop("cluster appears as both an argument and a model term")
    if (length(temp$vars) > 1) stop("can not have two cluster terms")
    cluster <- mf[[temp$vars]]
    Terms <- Terms[-temp$terms]
```

```
}

ll <- attr(Terms, 'term.labels')
if (length(ll) == 0) X <- factor(rep(1,n))  # ~1 on the right
else X <- strata(mf[ll])

# Backwards support for the now-depreciated etype argument
etype <- model.extract(mf, "etype")
if (!is.null(etype)) {
    if (attr(Y, "type") == "mcounting" ||
        attr(Y, "type") == "mright")
        stop("cannot use both the etype argument and mstate survival type")
    if (length(istate))
        stop("cannot use both the etype and istate arguments")
    status <- Y[,ncol(Y)]
    etype <- as.factor(etype)
    temp <- table(etype, status==0)

    if (all(rowSums(temp==0) ==1)) {
        # The user had a unique level of etype for the censors
        newlev <- levels(etype)[order(-temp[,2])] #censors first
    }
    else newlev <- c(" ", levels(etype)[temp[,1] >0])
    status <- factor(ifelse(status==0,0, as.numeric(etype)),
                         labels=newlev)

    if (attr(Y, 'type') == "right")
        Y <- Surv(Y[,1], status, type="mstate")
    else if (attr(Y, "type") == "counting")
        Y <- Surv(Y[,1], Y[,2], status, type="mstate")
    else stop("etype argument incompatable with survival type")
}
```

Once upon a time I allowed survfit to be called without the '~1' portion of the formula. This was a mistake for multiple reasons, but the biggest problem is timing. If the subject has a data statement but the first argument is not a formula, R needs to evaluate Surv(t,s) to know that it is a survival object, but it also needs to know that this is a survival object before evaluation in order to dispatch the correct method. The method below helps give a useful error message in some cases.

```
⟨survfit-Surv⟩=
survfit.Surv <- function(formula, ...)
    stop("the survfit function requires a formula as its first argument")
```

The last peice in this file is the function to create confidence intervals. It is called from multiple different places so it is well to have one copy. If p is the survival probability and $s(p)$ its standard error, we can do confidence intervals on the simple scale of $p \pm 1.96s(p)$, but that does not have very good properties. Instead use a transformation $y = f(p)$ for which the standard error is $s(p)f'(p)$, leading to the confidence interval

$$f^{-1}\left(f(p) + -1.96 s(p) f'(p)\right)$$

Here are the supported transformations.

	f	f'	f^{-1}
log	$\log(p)$	$1/p$	$\exp(y)$
log-log	$\log(-\log(p))$	$1/\left[p\log(p)\right]$	$\exp(-\exp(y))$
logit	$\log(p/1-p)$	$1/[p(1-p)]$	$1 - 1/\left[1+\exp(y)\right]$
arcsin	$\arcsin(\sqrt{p})$	$1/(2\sqrt{p(1-p)})$	$\sin^2(y)$

Plain intervals can give limits outside of (0,1), we truncate them when this happens. The log intervals can give an upper limit greater than 1, but the lower limit is always valid, and the log-log and logit. The arcsin require truncation in the middle of the formula. In all cases we return NA as the CI for survival=0: it makes the graphs look better.

Some of the underlying routines compute the standard error of p and some the standard error of $\log(p)$. The `selow` argument is used for the modified lower limits of Dory and Korn. When this is used for cumulative hazards the ulimit arg will be FALSE: no upper limit of 1.

```
⟨survfit⟩=
 survfit_confint <- function(p, se, logse=TRUE, conf.type, conf.int,
                             selow, ulimit=TRUE) {
     zval <- qnorm(1- (1-conf.int)/2, 0,1)
     if (missing(selow)) scale <- 1.0
     else scale <- ifelse(selow==0, 1.0, selow/se)  # avoid 0/0 at the origin
     if (!logse) se <- ifelse(se==0, 0, se/p)   # se of log(survival) = log(p)

     if (conf.type=='plain') {
         se2 <- se* p * zval  # matches equation 4.3.1 in Klein & Moeschberger
         if (ulimit) list(lower= pmax(p -se2*scale, 0), upper = pmin(p + se2, 1))
         else  list(lower= pmax(p -se2*scale, 0), upper = p + se2)
     }
     else if (conf.type=='log') {
         #avoid some "log(0)" messages
         xx <- ifelse(p==0, NA, p)
         se2 <- zval* se
         temp1 <- exp(log(xx) - se2*scale)
         temp2 <- exp(log(xx) + se2)
         if (ulimit) list(lower= temp1, upper= pmin(temp2, 1))
         else  list(lower= temp1, upper= temp2)
     }
     else if (conf.type=='log-log') {
         xx <- ifelse(p==0 | p==1, NA, p)
         se2 <- zval * se/log(xx)
         temp1 <- exp(-exp(log(-log(xx)) - se2*scale))
         temp2 <- exp(-exp(log(-log(xx)) + se2))
         list(lower = temp1 , upper = temp2)
     }
     else if (conf.type=='logit') {
         xx <- ifelse(p==0, NA, p)  # avoid log(0) messages
         se2 <- zval * se *(1 + xx/(1-xx))

         temp1 <- 1- 1/(1+exp(log(p/(1-p)) - se2*scale))
         temp2 <- 1- 1/(1+exp(log(p/(1-p)) + se2))
         list(lower = temp1, upper=temp2)
     }
     else if (conf.type=="arcsin") {
         xx <- ifelse(p==0, NA, p)
         se2 <- .5 *zval*se * sqrt(xx/(1-xx))
         list(lower= (sin(pmax(0, asin(sqrt(xx)) - se2*scale)))^2,
              upper= (sin(pmin(pi/2, asin(sqrt(xx)) + se2)))^2)
     }
     else stop("invalid conf.int type")
 }
```

9.2 Kaplan-Meier

This routine has been rewritten more times than any other in the package, as we trade off simplicty of the code with execution speed. This version does all of the oranizational work in S and calls a C routine for each separate curve. The first code did everything in C but was too hard to maintain and the most recent prior function did nearly everything in S. Introduction of robust variance prompted a movement of more of the code into C since that calculation is computationally intensive.

```
⟨survfitKM⟩=
 survfitKM <- function(x, y, weights=rep(1.0,length(x)),
                       stype=1, ctype=1,
                       se.fit=TRUE,
                       conf.int= .95,
                       conf.type=c('log',  'log-log',  'plain', 'none',
                                   'logit', "arcsin"),
                       conf.lower=c('usual', 'peto', 'modified'),
                       start.time, id, cluster, robust, influence=FALSE,
                       type) {

     if (!missing(type)) {
         if (!is.character(type)) stop("type argument must be character")
         # older style argument is allowed
         temp <- charmatch(type, c("kaplan-meier", "fleming-harrington", "fh2"))
         if (is.na(temp)) stop("invalid value for 'type'")
         type <- c(1,3,4)[temp]
```

```
}
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")
    type <- as.integer(2*stype + ctype  -2)
}

conf.type <- match.arg(conf.type)
conf.lower<- match.arg(conf.lower)
if (is.logical(conf.int)) {
    # A common error is for users to use "conf.int = FALSE"
    #  it's not correct, but allow it
    if (!conf.int) conf.type <- "none"
    conf.int <- .95
}

if (!is.Surv(y)) stop("y must be a Surv object")
if (attr(y, 'type') != 'right' && attr(y, 'type') != 'counting')
        stop("Can only handle right censored or counting data")
ny <- ncol(y)       # Will be 2 for right censored, 3 for counting
# The calling routine has used 'strata' on x, so it is a factor with
#  no unused levels.  But just in case a user called this...
if (!is.factor(x)) stop("x must be a factor")
xlev <- levels(x)   # Will supply names for the curves
x <- as.integer(x)  # keep the integer index

if (missing(start.time)) time0 <- min(0, y[,ny-1])
else time0 <- start.time

# 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<-      (!missing(weights) && any(weights != floor(weights)))
#has.rwt <- FALSE   # we are rethinking this
has.robust <-  !missing(robust) && !is.null(robust)
if (has.id) id <- as.factor(id)

if (missing(robust) || is.null(robust)) {
    if (influence) {
        robust <- TRUE
        if (!(has.cluster || has.id)) {
            cluster <- seq(along=x)
            has.cluster <- TRUE
        }
    }
    else if (has.cluster || has.rwt ||
             (has.id && 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) {
    if (has.id) {
        # treat the id as both identifier and clustering
        clname <- levels(id)
        cluster <- as.integer(id)
        ncluster <- length(clname)
    }
    else if (ncol(y)==2 || !has.robust) {
        # create our own clustering
        n <- nrow(y)
        cluster <- 1:n
        ncluster <- n
        clname <- 1:n
    }
    else stop("id or cluster option required")
} else ncluster <- 0

if (is.logical(influence)) {
    # TRUE/FALSE is treated as all or nothing
    if (!influence) influence <- 0L
    else influence <- 3L
}
else if (!is.numeric(influence))
    stop("influence argument must be numeric or logical")
if (!(influence %in% 0:3)) stop("influence argument must be 0, 1, 2, or 3")
else influence <- as.integer(influence)
if (!robust && influence >0) {
    warning("robust=FALSE implies influence=FALSE")
    influence <- 0L
}

if (!se.fit) {
    # if the user asked for no standard error, skip any robust computation
    ncluster <- 0L
    influence <- 0L
}

# if start.time was set, delete obs if necessary
keep <- y[,ny-1] >= time0
if (!all(keep)) {
    y <- y[keep,]
    if (length(id) >0) id <- id[keep]
    if (length(cluster) >0) cluster <- cluster[keep]
    x <- x[keep]
    weights <- weights[keep]
}


⟨survfitKM-compute⟩
⟨survfitKM-finish⟩
}
```

At each event time we have

- n(t) = number at risk = sum of weigths for those at risk
- d(t) = number of events = sum of weights for the deaths
- e(t) = unweighted number of events

From this we can calculate the Kapan-Meier and Nelson-Aalen estimates. The Fleming-Harrington estimate is the analog of the Efron approximation in a Cox model. When there are no case weights the FH idea is quite simple. Assume that the real data is not tied, but we saw a coarsened version. If we see 3 events out of 10 subjects at risk the NA increment is 3/10 but the FH is 1/10 + 1/9 + 1/8, it is what we would have seen with the uncoarsened data. If there are case weights we give each of the 3 terms a 1/3 chance of being the first, second, or third event

$$\begin{aligned}
KM(t) &= KM(t-)(1 - d(t)/n(t) \\
NA(t) &= NA(t-) + d(t)/n(t) \\
FH(t) &= FH(t-) + \sum_{i=1}^{3} \frac{(d(t)/3}{n(t) - d(t)(i-1)/3}
\end{aligned}$$

When one of these 3 subjects has an event but continues, which can happen with start/stop data, then this gets trickier: the second d in the last equation above should include only the other 2. The idea is that each of those will certainly be present for the first event, has 2/3 chance of being present for the second, and 1/3 for the third. If we think of the size of the denominator as a random variable Z, an exact solution would use $E(1/Z)$, the FH uses $1/E(Z)$ and the NA uses $1/\max(Z)$ as the denominator for each of the 3 deaths.

One problem with survival is near ties in Y: table, unique, ==, etc. can do different things in this case. Luckily, the parent survfit routine has dealt with that by using the `aeqSurv` function.

The underlying C code allows the sort1/sort2 vectors to be a different length than y, weights, and cluster. When there is only one curve we use that to our advantage to avoid creating a new copy of the last 3, passing in the original data. When there are multiple curves I had an internal debate about efficiency. Is is better to make a subset of y for each curve = more memory, or keep the original y and address a different subset in each C call = worse memory cache performance? I don't know the answer. In either case the cluster vector needs to be re-done for each group. Say that curve 1 uses subjects 1-10 and curve 2 uses 11-n: we don't want the first curve to compute or keep the zero influence values for all the subjects who are not in it. Especially when returning the influence matrix, which can get too large for memory.

If ny==3 and has.id is true, then do some extra setup work, which is to create a position vector of 1=first obs for the subject, 2 = last, 3=both, 0= other, for each set of back to back times. This is used to prevent counting a subject with data of (0,10], (10,15] in both the censored at 10 and entered at 10 totals. We assume the data has been vetted to prevent overlapping intervals, so that it suffices to sort by ending time. If a subject has holes in their timeline they will have more than one first and last indicator.

```
⟨survfitKM-compute⟩=
 if (ny==3 & has.id) position <- survflag(y, id)
 else position <- integer(0)

 if (length(xlev) ==1) {# only one group
     if (ny==2) {
         sort1 <- NULL
         sort2 <- order(y[,1])
     }
     else {
         sort2 <- order(y[,2])
         sort1 <- order(y[,1])
     }
     toss <- (y[sort2, ny-1] < time0)
```

```
    if (any(toss)) {
        # Some obs were removed by the start.time argument
        sort2 <- sort2[!toss]
        if (ny ==3) {
            index <- match(which(toss), sort1)
            sort1 <- sort1[-index]
        }
    }
    n.used <- length(sort2)
    if (ncluster > 0)
        cfit <- .Call(Csurvfitkm, y, weights, sort1-1L, sort2-1L, type,
                               cluster-1L, ncluster, position, influence)
    else cfit <- .Call(Csurvfitkm, y, weights, sort1-1L, sort2-1L, type,
                              0L, 0L, position, influence)
} else {
    # multiple groups
    ngroup <- length(xlev)
    cfit <- vector("list", ngroup)
    n.used <- integer(ngroup)
    if (influence) clusterid <- cfit # empty list of group id values
    for (i in 1:ngroup) {
        keep <- which(x==i & y[,ny-1] >= time0)
        if (length(keep) ==0) next;  # rare case where all are < start.time
        ytemp <- y[keep,]
        n.used[i] <- nrow(ytemp)
        if (ny==2) {
            sort1 <- NULL
            sort2 <- order(ytemp[,1])
        }
        else {
            sort2 <- order(ytemp[,2])
            sort1 <- order(ytemp[,1])
        }

        # Cluster is a nuisance: every curve might have a different set
        #  We need to relabel them from 1 to "number of unique clusters in this
        #  curve for the C routine
        if (ncluster > 0) {
            c2 <- cluster[keep]
            c.unique <- sort(unique(c2))
            nc <- length(c.unique)
            c2 <- match(c2, c.unique)  # renumber them
            if (influence >0) {
                clusterid[[i]] <-c.unique
            }
        }
        if (ncluster > 0)
            cfit[[i]] <- .Call(Csurvfitkm, ytemp, weights[keep], sort1 -1L,
                           sort2 -1L, type,
                           c2 -1L, length(c.unique), position, influence)
        else cfit[[i]] <- .Call(Csurvfitkm, ytemp, weights[keep], sort1 -1L,
                           sort2 -1L, type,
                           0L, 0L, position, influence)
    }
}
```

⟨*survfitKM-finish*⟩=

```
# create the survfit object
if (length(n.used) == 1) {
    rval <- list(n= length(x),
                 time= cfit$time,
                 n.risk = cfit$n[,4],
                 n.event= cfit$n[,5],
                 n.censor=cfit$n[,6],
                 surv = cfit$estimate[,1],
                 std.err = cfit$std.err[,1],
                 cumhaz  = cfit$estimate[,2],
                 std.chaz = cfit$std.err[,2])
 } else {
     strata <- sapply(cfit, function(x) if (is.null(x$n)) 0L else nrow(x$n))
     names(strata) <- xlev
     # we need to collapse the curves
     rval <- list(n= as.vector(table(x)),
                  time =   unlist(lapply(cfit, function(x) x$time)),
                  n.risk=  unlist(lapply(cfit, function(x) x$n[,4])),
                  n.event= unlist(lapply(cfit, function(x) x$n[,5])),
                  n.censor=unlist(lapply(cfit, function(x) x$n[,6])),
                  surv =   unlist(lapply(cfit, function(x) x$estimate[,1])),
                  std.err =unlist(lapply(cfit, function(x) x$std.err[,1])),
                  cumhaz  =unlist(lapply(cfit, function(x) x$estimate[,2])),
                  std.chaz=unlist(lapply(cfit, function(x) x$std.err[,2])),
                  strata=strata)
      if (ny==3) rval$n.enter <- unlist(lapply(cfit, function(x) x$n[,8]))
}

if (ny ==3) {
        rval$n.enter <- cfit$n[,8]
        rval$type <- "counting"
}
else rval$type <- "right"
```

```
if (se.fit) {
    rval$logse = (ncluster==0 || (type==2 || type==4))  # se(log S) or se(S)
    rval$conf.int   = conf.int
    rval$conf.type= conf.type
    if (conf.lower != "usual") rval$conf.lower = conf.lower

    if (conf.lower == "modified") {
        nstrat = length(n.used)
        events <- rval$n.event >0
        if (nstrat ==1) events[1] <- TRUE
        else            events[1 + cumsum(c(0, rval$strata[-nstrat]))] <- TRUE
        zz <- 1:length(events)
        n.lag <- rep(rval$n.risk[events], diff(c(zz[events], 1+max(zz))))
        #
        # n.lag = the # at risk the last time there was an event (or
        #   the first time of a strata)
        #
    }
    std.low <- switch(conf.lower,
                      ’usual’ = rval$std.err,
                      ’peto’ = sqrt((1-rval$surv)/ rval$n.risk),
                      ’modified’ = rval$std.err * sqrt(n.lag/rval$n.risk))

    if (conf.type != "none") {
        ci <- survfit_confint(rval$surv, rval$std.err, logse=rval$logse,
                              conf.type, conf.int, std.low)
        rval <- c(rval, list(lower=ci$lower, upper=ci$upper))
     }
} else {
    # for consistency don’t return the se if std.err=FALSE
    rval$std.err <- NULL
    rval$std.chaz <- NULL
}

# Add the influence, if requested by the user
#  remember, if type= 3 or 4, the survival influence has to be constructed.
if (influence > 0) {
    if (type==1 | type==2) {
        if (influence==1 || influence ==3) {
            if (length(xlev)==1) {
                rval$influence.surv <- cfit$influence1
                row.names(rval$influence.surv) <- clname
            }
            else {
                temp <- vector("list", ngroup)
                for (i in 1:ngroup) {
```

```
                temp[[i]] <- cfit[[i]]$influence1
                row.names(temp[[i]]) <- clname[clusterid[[i]]]
            }
            rval$influence.surv <- temp
        }
    }
    if (influence==2 || influence==3) {
        if (length(xlev)==1) {
            rval$influence.chaz <- cfit$influence2
            row.names(rval$influence.chaz) <- clname
        }
        else {
            temp <- vector("list", ngroup)
            for (i in 1:ngroup) {
                temp[[i]] <- cfit[[i]]$influence2
                row.names(temp[[i]]) <- clname[clusterid[[i]]]
            }
            rval$influence.chaz <- temp
        }
    }
}
else {
    # everything is derived from the influence of the cumulative hazard
    if (length(xlev) ==1) {
        temp <- cfit$influence2
        row.names(temp) <- clname
    } else {
        temp <- vector("list", ngroup)
        for (i in 1:ngroup) {
            temp[[i]] <- cfit[[i]]$influence2
            row.names(temp[[i]]) <- clname[clusterid[[i]]]
        }
    }

    if (influence==2 || influence ==3)
        rval$influence.chaz <- temp

    if (influence==1 || influence==3) {
        # if an obs moves the cumulative hazard up, then it moves S down
        if (length(xlev) ==1)
            rval$influence.surv <- -temp * rep(rval$surv, each=nrow(temp))
        else {
            for (i in 1:ngroup)
                temp[[i]] <- -temp[[i]] * rep(cfit[[i]]$estimate[,1],
                                             each=nrow(temp[[i]]))
            rval$influence.surv <- temp
            }
        }
    }
}

if (!missing(start.time)) rval$start.time <- start.time
rval
```

Now for the real work using C routines. My standard for a variable named “zed” is to use zed2 for the S object and zed for the data part of the object; the latter is what the C code works with.

⟨*survfitkm*⟩=

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

SEXP survfitkm(SEXP y2, SEXP weight2,  SEXP sort12, SEXP sort22,
               SEXP type2, SEXP id2, SEXP nid2,   SEXP position2,
               SEXP influence2) {

    int i, i1, i2, j, k, person1, person2;
    int nused, nid, type, influence;
    int ny, ntime;
    double *tstart=0, *stime, *status, *wt;
    double v1, v2, dtemp, haz;
    double temp, dtemp2, dtemp3, frac, btemp;
    int *sort1=0, *sort2, *id=0;
    static const char *outnames[]={"time", "n", "estimate", "std.err",
                                     "influence1", "influence2", ""};
    SEXP rlist;
    double *gwt=0, *inf1=0, *inf2=0;  /* =0 to silence -Wall */
    int *gcount=0;
    int n1, n2, n3, n4;
    int *position=0, hasid;
    double wt1, wt2, wt3, wt4;

    /* output variables */
    double  *n[10],  *dtime,
           *kvec, *nvec, *std[2], *imat1=0, *imat2=0; /* =0 to silence -Wall*/
    double km, nelson;  /* current estimates */

    /* map the input data */
    ny = ncols(y2);     /* 2= ordinary survival 3= start,stop data */
    nused = nrows(y2);
    if (ny==3) {
        tstart = REAL(y2);
        stime = tstart + nused;
        sort1 = INTEGER(sort12);
    }
    else stime = REAL(y2);
    status= stime +nused;
    wt = REAL(weight2);
    sort2 = INTEGER(sort22);
    nused = LENGTH(sort22);

    type = asInteger(type2);
    nid = asInteger(nid2);
    if (LENGTH(position2) > 0) {
        hasid =1;
        position = INTEGER(position2);
    } else hasid=0;
    influence = asInteger(influence2);

    /* nused was used for two things just above.  The first was the length of
       the input data y, only needed for a moment to set up tstart, stime, and
       status.  The second is the number of these observations we will actually
       use, which is the length of sort2.  This routine can be called multiple
       times with sort1/sort2 pointing to different subsets of the data while
       y, wt, id and position can remain unchanged
    */

    if (length(id2)==0) nid =0;  /* no robust variance */
    else id = INTEGER(id2);

    /* pass 1, get the number of unique times, needed for memory allocation
      Number of xval groups (unique id values) has been supplied
      Data is sorted by time
    */
    ntime =1;
    temp = stime[sort2[0]];
    for (i=1; i<nused; i++) {
        i2 = sort2[i];
        if (stime[i2] != temp) {
            ntime++;
            temp = stime[i2];
        }
    }
```

I don't want to rewrite this routine again, so make the C code return most everything that I think I will ever want. For type=1,2 compute the KM, otherwise use exp(chaz). For type=1,3 compute the Nelson-Aalen and for 2,4 the Efron variant of the NA as the cumulative hazard. Without robust variance, do the simple Greenwood and NA variances. Robust variances take more time since they are by their nature $O(gm)$ where g is the number of groups and m the number of unique times. If type= 3,4 we only compute the robust verion of the cumulative hazard variance; the influence matrix for the survival in this case is $-\exp(-H(t))D(t)$ where H is the cumulative hazard and D is the influence matrix for H. This can be done in the R code. The influence matrices are large, so only return what they explicitly ask for.

```
⟨survfitkm⟩=
    /* Allocate memory for the output
        n has 6 columns for number at risk, events, censor, then the
        3 weighted versions of the same, then optionally two more for
        number added to the risk set (when ny=3)
    */
    PROTECT(rlist = mkNamed(VECSXP, outnames));

    dtime  = REAL(SET_VECTOR_ELT(rlist, 0, allocVector(REALSXP, ntime)));
    if (ny==2) j=7;  else j=9;
    n[0]  = REAL(SET_VECTOR_ELT(rlist, 1, allocMatrix(REALSXP, ntime, j)));
    for (i=1; i<j; i++) n[i] = n[0] + i*ntime;

    kvec  = REAL(SET_VECTOR_ELT(rlist, 2, allocMatrix(REALSXP, ntime, 2)));
    nvec  = kvec + ntime;  /* Nelson-Aalen estimate */
    std[0] = REAL(SET_VECTOR_ELT(rlist, 3, allocMatrix(REALSXP, ntime,2)));
    std[1] = std[0] + ntime;

    if (nid >0 ) { /* robust variance */
        gcount = (int *) R_alloc(nid, sizeof(int));
        if (type <3) {  /* working vectors for the influence */
            gwt  = (double *) R_alloc(3*nid, sizeof(double));
            inf1 = gwt + nid;
            inf2 = inf1 + nid;
            for (i=0; i< nid; i++) {
                gwt[i] =0.0;
                gcount[i] = 0;
                inf1[i] =0;
                inf2[i] =0;
            }
        }
        else {
            gwt = (double *) R_alloc(2*nid, sizeof(double));
            inf2 = gwt + nid;
            for (i=0; i< nid; i++) {
                gwt[i] =0.0;
                gcount[i] = 0;
                inf2[i] =0;
            }
        }
        /* these are not accumulated, so do not need to be zeroed */
        if (type <3) {
            if (influence==1 || influence ==3)
                imat1 = REAL(SET_VECTOR_ELT(rlist, 4,
                                     allocMatrix(REALSXP, nid, ntime)));
            if (influence==2 || influence==3)
                imat2 =  REAL(SET_VECTOR_ELT(rlist, 5,
                           allocMatrix(REALSXP, nid, ntime)));
        }
        else  if (influence !=0)
                imat2 = REAL(SET_VECTOR_ELT(rlist, 5,
                               allocMatrix(REALSXP, nid, ntime)));
    }

    ⟨survfitkm-pass2⟩
    ⟨survfitkm-pass3⟩

    UNPROTECT(1);
    return(rlist);
}
```

Pass 2 goes from the last time to the first and fills in the `n` matrix.

```
⟨survfitkm-pass2⟩=
R_CheckUserInterrupt();  /*check for control-C */
/*
** person1, person2 track through sort1 and sort2, respectively
**  likewise with i1 and i2
*/
person1 = nused-1;  person2 = nused-1;
n1=0; wt1=0;
for (k=ntime-1; k>=0; k--) {
    dtime[k] = stime[sort2[person2]];  /* current time point */
    n2=0; n3=0; wt2=0; wt3=0;
    for (; person2 >=0; person2--) {
        i2= sort2[person2];
        if (stime[i2] != dtime[k]) break;

        n1++;             /* number at risk */
        wt1 += wt[i2];    /* weighted number at risk */
        if (status[i2] ==1) {
            n2++;           /* events */
            wt2 += wt[i2];
        } else if (hasid==0 || (position[i2]& 2)) {
            /* if there are no repeated obs for a subject (hasid=0)
            **  or this is the last of a string (a,b](b,c](c,d].. for
            **  a subject (position[i2]=2 or 3), then it is a 'real' censor
            */
            n3++;
            wt3 += wt[i2];
        }
    }

    if (ny==3) { /* remove any with start time >=dtime*/
        n4 =0; wt4 =0;
        for (; person1 >=0; person1--) {
            i1 = sort1[person1];
            if (tstart[i1] < dtime[k]) break;
            n1--;
            wt1 -= wt[i1];
            if (hasid==0 || (position[i1] & 1)) {
                /* if there are no repeated id (hasid=0) or this is the
                ** first of a string of (a,b](b,c](c,d] for a subject, then
                ** this is a 'real' entry */
                n4++;
                wt4 += wt[i1];
            }
        }
        if (n4>0) {
           n[6][k+1] = n4;
           n[7][k+1] = wt4;
       }
    }

    n[0][k] = n1;  n[1][k]=n2;  n[2][k]=n3;
    n[3][k] = wt1; n[4][k]=wt2; n[5][k]=wt3;
}

if (ny ==3) {   /* fill in number entered for the initial interval */
    n4=0; wt4=0;
    for (; person1>=0; person1--) {
        i1 = sort1[person1];
        if (hasid==0 || (position[i1] & 1)) {
            n4++;
            wt4 += wt[i1];
        }
    }
    n[6][0] = n4;
    n[7][0] = wt4;
}
```

The rest of the code is identical for simple survival or start-stop data. The cumulative hazard estimates are the Nelson-Aalen-Breslow (same estimate, three different papers) or the Fleming-Harrington.

$$\Lambda_A(t) \sum u_j \le t d_j/r_j$$
$$\Lambda_{FH}(t) = \sum u_j \le t \frac{d_j}{f_j \sum_{k=0}^{f_j-1}(r_j - kd_j/f_j)}$$

To understand the Fleming-Harrington estimate, suppose that at some time point we had three deaths out of 10 at risk. The Aalen estimate gives a hazard estimate of 3/10. The FH estimate assumes that the deaths didn't actually all happen at once, even though rounding in the data collection process makes it appear that way, so the better estimate is 1/10 + 1/9 + 1/8. The third person to die, whoever that was, would have had only 8 at risk when their event happened.

The estimate of survival is either the Kaplan-Meier or the exponential of the hazard.

$$KM(t) = \prod_{u_j \le t} \frac{r_j - d_j}{r_j}$$

The third pass goes from smallest time to largest. 99 times out of 100 the user will choose type=1, so we try to avoid testing those expression n times.

⟨*survfitkm-pass3*⟩=
```
R_CheckUserInterrupt();  /*check for control-C */
nelson =0.0; km=1.0;
v1=0; v2=0;
if (nid==0) {  /* simple variance */
   if (type==1 || type==3) {  /* Nelson-Aalen hazard */
        for (i=0; i<ntime; i++) {
            if (n[1][i]>0 && n[4][i]>0) {  /* at least one event with wt>0*/
                nelson += n[4][i]/n[3][i];
                v2 += n[4][i]/(n[3][i]*n[3][i]);
                }

            nvec[i] = nelson;
            std[0][i] = sqrt(v2);
            std[1][i] = sqrt(v2);
            }
   } else {              /* Fleming hazard */
        for (i=0; i<ntime; i++) {
            for (j=0; j<n[1][i]; j++) {
                dtemp = n[3][i] - j*n[4][i]/n[1][i];
                nelson += n[4][i] /(n[1][i]* dtemp);
                v2 += n[4][i]/(n[1][i]*dtemp*dtemp);
            }
            kvec[i] = exp(-nelson);
            nvec[i] = nelson;
            std[0][i] = sqrt(v2);
            std[1][i] = sqrt(v2);
        }
    }

    if (type < 3) {  /* KM survival */
        for (i=0; i<ntime; i++) {
            if (n[1][i]>0 && n[4][i]>0) {  /* at least one event */
                km *= (n[3][i]-n[4][i])/n[3][i];
                v1 += n[4][i]/(n[3][i] * (n[3][i] - n[4][i])); /* Greenwood */
                }
            kvec[i] = km;
            std[0][i] = sqrt(v1);
        }
    } else {  /* exp survival */
        for (i=0; i< ntime; i++) {
            kvec[i] = exp(-nvec[i]);
            std[0][i] = std[1][i];
        }
    }
}

else { /* infinitesimal jackknife variance */
    ⟨survfitkm-influence⟩
}
```

The robust variance is based on an infinitesimal jackknife (IJ). Let $S_{-i}(t)$ be the survival curve without subject i and $J_i(t) = S_i(t) - S_{-i}(t)$ be the change in the survival curve from adding subject i back in. Then the jackknife estimate of variance is

$$\sigma^2_J(t) = \sum \left(J_i(t) - \overline{J}(t)\right)^2$$

The IJ estimate instead uses the linear approximation to the jackknife, since it is normally less work to compute the derivative than a whole new estimate. Notice that if all the weights were doubled the expression below will stay the same since the derivative will drop by 1/2.

$$\sigma^2_{IJ}(t_k) = \sum_i w_i U^2_{ik}$$
$$U_{ik} = \frac{\partial S(t_k)}{\partial w_i}$$

The big problem with the IJ estimate is that a first derivative matrix U will have one row per subject and one column per event time. Since the number of unique event times tends to grow with n, this matrix very rapidly becomes too large to manage. Instead use a grouped jackknife with g groups, g will often be on the order of 20–50. The 0/1 design matrix B has n rows and g columns, one column per group, marking which subject is in each group. The grouped jackknife can be written as

$$U'WBB'WU = V'V$$

Our goal is to accumulate and use V instead of U. The working vectors `inf1` and `inf2` contain the current estimate for the survival S and cumulative hazard H, at a given time. They are saved into the `imat` array for users, if desired.

First work this out for the cumulative hazard, which is simpler, and a single subject k.

$$\begin{aligned}
H(t) &= \sum_{s \le t} \frac{\sum w_i dN_i(s)}{\sum w_i Y_i(s)} \\
&= \sum_{s \le t} h(s) \\
U_k(t) &= U_k(t-) + \frac{\partial h(t)}{\partial w_k} \\
&= U_k(t-) + \frac{1}{\sum w_i Y_i(t)} \left(dN_k(t) - Y_k(t)h(t)\right) \qquad (25) \\
\sum_k w_k U_k(t) &= 0
\end{aligned}$$

using the counting process notation of $N(t)$ for events and $Y(t)$ for at risk. The weighted sum of the first derivatives is zero, so we don't need a mean when computing the variance estimate. (This is true for all IJ estimators.) V involves the weighted sum of this over groups, for the increment to each row of V the rightmost term of (25) is replaced by the weighted sum over each group. Since h is the same for every subject at risk, we only need accumulate the sum of subjects in and sum of events in each group. The first can be kept as a running sum with $O(n)$ effort.

When using the FH2 estimate tied deaths are different. Say that subject i dies at some time t where there are 2 other tied deaths. Let w_i for $i = 1, 2, 3$ be the weight of those who die and s the sum of weights for all the others. The contribution to the cumulative hazard and derivative at this time point is

$$\begin{aligned}
h &= \frac{w_1 + w_2 + w_3}{3} \frac{1}{s + w_1 + w_2 + w_3} + \frac{w_1 + w_2 + w_3}{3} \frac{1}{s + 2(w_1 + w_2 + w_3)/3} + \frac{w_1 + w_2 + w_3}{3} \frac{1}{s + (w_1 + w_2 + w_3)/3} \\
&\equiv a(b_1 + b_2 + b_3) \frac{\partial h}{\partial w_i}
\end{aligned}$$

The idea is that if the data had been gathered with more precision, then there would not be ties. The first death has 1/3 chance of being subject 1,2, or 3 and all are in the denominator. The second also has 1/3 chance of being 1–3, and each of these has 2/3 chance of still being in the denominator, etc. The standard variance will be $ab_1^2 + ab_2^2 + ab_3^2$.

For the Kaplan-Meier we have

$$\begin{aligned}
KM(t) &= KM(t-)[1 - h(t)] \\
U_k(t) &= \frac{\partial KM(t)}{\partial w_k} \\
&= U_k(t-)[1 - h(t)] - KM(t-) \frac{\partial h(t)}{\partial w_k} \qquad (26)
\end{aligned}$$

The V matrix is again a weighted sum. The first term of (26) does not change, it multiplies the current value times $1 - h$. The second term involves the same summation as the cumulative hazard.

When using $\exp(-H)$ as the survival estimate then

$$\begin{aligned}\frac{partial S(t)}{\partial w_k} &= \frac{\partial e^{-H(t)}}{\partial w_k} \\ &= e^{-H(t)} \partial H(t) \partial w_k\end{aligned}$$

so in this case only the robust variance for the cumulative hazard H is needed, and the parent R routine can fill in the rest.

The variance for a given survival time is $\sum V^2$, which is always returned. The code keeps the current V vector for the hazard H in `inf2`, and if necessary that for the KM in `inf1`. A last step is to add up the squares of all of these, so the algorithm is $O(gp)$ where p is the number of unique event times and g is the number of groups. The sum of weights for each group is kept in a vector `gwt`, which is updated as subjects enter and leave.

Say that a study had n= 10 million subjects in g=100 groups with d = 1 million deaths. At each death we update the 'hazard' part of the influence for all 100 groups, which is O(gd). The deaths at that time point have a second increment, to whichever group each is in, but since time is sorted that adds O(n) for indexing and O(d) for the work. The most important thing is to avoid doing anything that would be O(ng) or O(nd). For method=3 the hazard part of the increment is also different for a death, the solution is to do an ordinary increment for everyone in the O(gd) step, then correct it when doing the O(d) update.

```
⟨survfitkm-influence⟩=
 v1=0; v2 =0; km=1; nelson =0;
 person2=0;
 if (ny==3) {
     person1 =0;
 } else {
     /* at the start, everyone is at risk */
     for (i=0; i< nused; i++) {
         i2 = id[i];
         gcount[i2]++;
         gwt[i2] += wt[i];
     }
 }

 if (type==1) {
     for (i=0; i< ntime; i++) {
         if (ny==3) {
             /* add in new subjects */
             for (; person1 < nused; person1++) {
                 /* add in those whose start time is < dtime */
                 i1 = sort1[person1];
                 if (tstart[i1] >= dtime[i]) break;
```

```
        gcount[id[i1]]++;
        gwt[id[i1]] += wt[i1];
    }
}

if (n[1][i] > 0 && n[4][i]>0) { /* need to update the sums */
    haz = n[4][i]/n[3][i];
    for (k=0; k< nid; k++) {
        inf1[k] = inf1[k] *(1.0 -haz) + gwt[k]*km*haz/n[3][i];
        inf2[k] -= gwt[k] * haz/n[3][i];
    }
    for (; person2<nused; person2++) {
        i2 = sort2[person2];
        if (stime[i2] > dtime[i]) break;   /* those at this time */
        if (status[i2]==1) {
            inf1[id[i2]] -= km* wt[i2]/n[3][i];
            inf2[id[i2]] += wt[i2]/n[3][i];
        }
        gcount[id[i2]] --;
        if (gcount[id[i2]] ==0) gwt[id[i2]] = 0.0;
        else gwt[id[i2]] -= wt[i2];
    }
    km *= (1-haz);
    nelson += haz;

    v1=0; v2=0;
    for (k=0; k<nid; k++) {
        v1 += inf1[k]*inf1[k];
        v2 += inf2[k]*inf2[k];
    }
} else {  /* only need to udpate weights */
    for (; person2<nused; person2++) {
        i2 = sort2[person2];
        if (stime[i2] > dtime[i]) break;
        gcount[id[i2]] --;
        if (gcount[id[i2]] ==0) gwt[id[i2]] = 0.0;
        else gwt[id[i2]] -= wt[i2];
    }
}

kvec[i] = km;
nvec[i] = nelson;
std[0][i] = sqrt(v1);
std[1][i] = sqrt(v2);
if (influence==1 || influence ==3)
    for (k=0; k<nid; k++) *imat1++ = inf1[k];
```

```
        if (influence==2 || influence ==3)
            for (k=0; k<nid; k++) *imat2++ = inf2[k];
    }
}
else if (type==2) {  /* KM survival, Fleming-Harrington hazard */
    for (i=0; i< ntime; i++) {
        if (ny==3) {
            /* add in new subjects */
            for (; person1<nused; person1++) {
                i1 = sort1[person1];
                if (tstart[i1] >= dtime[i]) break;
                gcount[id[i1]]++;
                gwt[id[i1]] += wt[i1];
            }
        }

        if (n[1][i] > 0 && n[4][i] >0) { /* need to update the sums */
            dtemp =0;  /* the working denominator */
            dtemp2=0;  /* sum of squares */
            dtemp3=0;
            temp = n[3][i] - n[4][i];  /* sum of weights for the non-deaths */
            for (k=n[1][i]; k>0; k--) {
                frac = k/n[1][i];
                btemp = 1/(temp + frac*n[4][i]);  /* "b" in the math */
                dtemp += btemp;
                dtemp2 += btemp*btemp*frac;
                dtemp3 += btemp*btemp;    /* non-death deriv */
            }

            dtemp /=  n[1][i];        /* average denominator */
            if (n[4][i] != n[1][i]) { /* case weights */
                dtemp2 *= n[4][i]/ n[1][i];
                dtemp3 *= n[4][i]/ n[1][i];
            }
            nelson += n[4][i]*dtemp;

            haz = n[4][i]/n[3][i];
            for (k=0; k< nid; k++) {
                inf1[k] = inf1[k] *(1.0 -haz) + gwt[k]*km*haz/n[3][i];
                if (gcount[k]>0) inf2[k] -= gwt[k] * dtemp3;
            }
            for (; person2<nused; person2++) {
                /* catch the endpoints up to this event time */
                i2 = sort2[person2];
                if (stime[i2] > dtime[i]) break;
                if (status[i2]==1) {
                    inf1[id[i2]] -= km* wt[i2]/n[3][i];
                    inf2[id[i2]] += wt[i2] *(dtemp + dtemp3 - dtemp2);
                 }
                gcount[id[i2]] --;
                if (gcount[id[i2]] ==0) gwt[id[i2]] = 0.0;
                else gwt[id[i2]] -= wt[i2];
            }
            km *= (1-haz);

            v1=0; v2=0;
            for (k=0; k<nid; k++) {
                v1 += inf1[k]*inf1[k];
                v2 += inf2[k]*inf2[k];
            }
        } else {  /* only need to udpate weights */
            for (; person2<nused; person2++) {
                i2 = sort2[person2];
                if (stime[i2] > dtime[i]) break;
                gcount[id[i2]] --;
                if (gcount[id[i2]] ==0) gwt[id[i2]] = 0.0;
                else gwt[id[i2]] -= wt[i2];
            }
        }

        kvec[i] = km;
        nvec[i] = nelson;
        std[0][i] = sqrt(v1);
        std[1][i] = sqrt(v2);
        if (influence==1 || influence ==3)
            for (k=0; k<nid; k++) *imat1++ = inf1[k];
        if (influence==2 || influence ==3)
            for (k=0; k<nid; k++) *imat2++ = inf2[k];
    }
}

else if (type==3) {  /* exp() survival, NA hazard */
    for (i=0; i< ntime; i++) {
        if (ny==3) {
            /* add in new subjects */
            for (; person1 < nused; person1++) {
                i1 = sort1[person1];
                if (tstart[i1] >= dtime[i]) break;

                gcount[id[i1]]++;
                gwt[id[i1]] += wt[i1];
            }
        }

        if (n[1][i] > 0 && n[4][i]>0) { /* need to update the sums */
            haz = n[4][i]/n[3][i];
            for (k=0; k< nid; k++) {
                inf2[k] -= gwt[k] * haz/n[3][i];
            }
            for (; person2<nused; person2++) {
                /* catch the endpoints up to this event time */
                i2 = sort2[person2];
                if (stime[i2] > dtime[i]) break;
                if (status[i2]==1) {
                     inf2[id[i2]] += wt[i2]/n[3][i];
                }
                gcount[id[i2]] --;
                if (gcount[id[i2]] ==0) gwt[id[i2]] = 0.0;
                else gwt[id[i2]] -= wt[i2];
            }
            nelson += haz;

            v2=0;
            for (k=0; k<nid; k++) {
                v2 += inf2[k]*inf2[k];
            }
        } else {  /* only need to udpate weights */
            for (; person2<nused; person2++) {
                i2 = sort2[person2];
                if (stime[i2] > dtime[i]) break;
                gcount[id[i2]] --;
                if (gcount[id[i2]] ==0) gwt[id[i2]] = 0.0;
                else gwt[id[i2]] -= wt[i2];
            }
        }

        kvec[i] = exp(-nelson);
        nvec[i] = nelson;
        std[1][i] = sqrt(v2);
        std[0][i] = sqrt(v2);

        if (influence>0)
            for (k=0; k<nid; k++) *imat2++ = inf2[k];
    }

} else {  /* exp() survival,  Fleming-Harrington hazard */
    for (i=0; i< ntime; i++) {
        if (ny==3) {
```

```
    /* add in new subjects */
    for (; person1 < nused; person1++) {
        i1 = sort1[person1];
        if (tstart[i1] >= dtime[i]) break;
        gcount[id[i1]]++;
        gwt[id[i1]] += wt[i1];
    }
}
if (n[1][i] > 0 && n[4][i] >0) { /* need to update the sums */
        dtemp =0;  /* the working denominator */
    dtemp2=0;  /* sum of squares */
    dtemp3=0;
    temp = n[3][i] - n[4][i];  /* sum of weights for the non-deaths */
    for (k=n[1][i]; k>0; k--) {
        frac = k/n[1][i];
        btemp = 1/(temp + frac*n[4][i]);  /* "b" in the math */
        dtemp += btemp;
        dtemp2 += btemp*btemp*frac;
        dtemp3 += btemp*btemp;    /* non-death deriv */
    }

    dtemp /=  n[1][i];        /* average denominator */
    if (n[4][i] != n[1][i]) { /* case weights */
        dtemp2 *= n[4][i]/ n[1][i];
        dtemp3 *= n[4][i]/ n[1][i];
    }
    nelson += n[4][i]*dtemp;

    for (k=0; k< nid; k++) {
        if (gcount[k]>0) inf2[k] -= gwt[k] * dtemp3;
    }
    for (; person2<nused; person2++) {
         i2 = sort2[person2];
        if (stime[i2] > dtime[i]) break;
        if (status[i2]==1) {
            inf2[id[i2]] += wt[i2] *(dtemp + dtemp3 - dtemp2);
        }
        gcount[id[i2]] --;
        if (gcount[id[i2]] ==0) gwt[id[i2]] = 0.0;
        else gwt[id[i2]] -= wt[i2];
    }

    v2=0;
    for (k=0; k<nid; k++) v2 += inf2[k]*inf2[k];
}
else { /* only need to update weights */
```

```
            for (; person2<nused; person2++) {
                i2 = sort2[person2];
                if (stime[i2] > dtime[i]) break;
                gcount[id[i2]] --;
                if (gcount[id[i2]] ==0) gwt[id[i2]] = 0.0;
                else gwt[id[i2]] -= wt[i2];
            }
        }

        kvec[i] = exp(-nelson);
        nvec[i] = nelson;
        std[1][i] = sqrt(v2);
        std[0][i] = sqrt(v2);

        if (influence>0)
            for (k=0; k<nid; k++) *imat2++ = inf2[k];
    }
}
```

9.3 Competing risks

The competing risks routine is very general, allowing subjects to enter or exit states multiple times. Early on I used the label *current prevalence* estimate, since it estimates what fraction of the subjects are in any given state across time. However the word “prevalence” is likely to generate confusion whenever death is one of the states, due to its historic use as the fraction of living subjects who have a particular condition. We will use the phrase *probability in state* or simply P from this point forward.

The easiest way to understand the estimate is to consider first the case of no censoring. In that setting the estimate of $p_k(t)$ for all states is obtained from a simple table of the current state at time t of the subjects, divided by n, the original sample size. When there is censoring the conceptually simple way to extend this is via the redistribute-to-the-right algorithm, which allocates the case weight for a censored subject evenly to all the others in the same state at the time of censoring.

The literature refers to these as “cumulative incidence” curves, which is confusing since P(state) is not the integral of incidence, but the routine name survfitCI endures. The cannonical call is one of

```
  fit <- survfit(Surv(time, status) ~ sex, data=mine)
  fit <- survfit(Surv(time1, time2, status) ~ sex, id= id, data=mine)
```

where `status` is a factor variable. Optionally, there can be an id statement or cluster term to indicate a data set with multiple transitions per subject. For multi-state survival the status variable has multiple levels, the first of which by default is censoring, and others indicating the type of transition that occured. The result will be a matrix of survival curves, one for each event type. If no initial state is specified then subjects are assumed to start in a "null" state, which gets listed last and by default will not be printed or plotted. (But it is present, with a name of ‘’);

The first part of the code is standard, parsing out options and checking the data.

```
⟨survfitCI⟩=
 ⟨survfitCI-compute⟩
 survfitCI <- function(X, Y, weights, id, cluster, robust, istate,
                       stype=1, ctype=1,
                       se.fit=TRUE,
                       conf.int= .95,
                       conf.type=c('log',  'log-log',  'plain', 'none',
                                   'logit', "arcsin"),
                       conf.lower=c('usual', 'peto', 'modified'),
                       influence = FALSE, start.time, p0, type){

     if (!missing(type)) {
         if (!missing(ctype) || !missing(stype))
             stop("cannot have both an old-style 'type' argument and the stype/ctype arguments that
         if (!is.character(type)) stop("type argument must be character")
         # older style argument is allowed
         temp <- charmatch(type, c("kaplan-meier", "fleming-harrington", "fh2"))
         if (is.na(temp)) stop("invalid value for 'type'")
         type <- c(1,3,4)[temp]
     }
     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")
         type <- as.integer(2*stype + ctype  -2)
     }
     if (type != 1) warning("only stype=1, ctype=1 currently implimented for multi-state data")

     conf.type <- match.arg(conf.type)
     conf.lower<- match.arg(conf.lower)
     if (conf.lower != "usual")
         warning("conf.lower is ignored for multi-state data")
     if (is.logical(conf.int)) {
         # A common error is for users to use "conf.int = FALSE"
         #  it's illegal per documentation, but be kind
         if (!conf.int) conf.type <- "none"
         conf.int <- .95
     }


     if (is.logical(influence)) {
         # TRUE/FALSE is treated as all or nothing
         if (!influence) influence <- 0L
         else influence <- 3L
     }
```

```
else if (!is.numeric(influence))
    stop("influence argument must be numeric or logical")
if (!(influence %in% 0:3)) stop("influence argument must be 0, 1, 2, or 3")
else influence <- as.integer(influence)

if (!se.fit) {
    # if the user asked for no standard error, skip any robust computation
    ncluster <- 0L
    influence <- 0L
}

type <- attr(Y, "type")
# This line should be unreachable, unless they call "surfitCI" directly
if (type !=’mright’ && type!=’mcounting’)
     stop(paste("multi-state computation doesn’t support \"", type,
                      "\" survival data", sep=’’))

# If there is a start.time directive, start by removing any prior events
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]
        weights <- weights[-toss]
        if (length(id) ==n) id <- id[-toss]
        if (!missing(istate) && length(istate)==n) istate <- istate[-toss]
        }
}
n <- nrow(Y)
status <- Y[,ncol(Y)]
ncurve <- length(levels(X))

# The user can call with cluster, id, robust or any combination.
# If only id, treat it as the cluster too
if (missing(robust) || length(robust)==0) robust <- TRUE
if (!robust) stop("multi-state survfit supports only a robust variance")

has.cluster <-  !(missing(cluster) || length(cluster)==0)
has.id <-       !(missing(id) || length(id)==0)
if (has.id) id <- as.factor(id)
else  {
        if (ncol(Y) ==3) stop("an id statement is required for start,stop data")
        id <- 1:n  # older default, which could lead to invalid curves
    }
    if (influence && !(has.cluster || has.id)) {
        cluster <- seq(along.with=X)
        has.cluster <- TRUE
    }

    if (has.cluster) {
        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 (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
            clname <- NULL
        }
    }

    if (missing(istate) || is.null(istate))
        mcheck <- survcheck2(Y, id)
    else mcheck <- survcheck2(Y, id, istate)
    if (any(mcheck$flag > 0)) stop("one or more flags are >0 in survcheck")
    states <- mcheck$states
    istate <- mcheck$istate
    nstate <- length(states)
    smap <- c(0, match(attr(Y, "states"), states))
    Y[,ncol(Y)] <- smap[Y[,ncol(Y)] +1]      # new states may be a superset
    status <- Y[,ncol(Y)]

    if (mcheck$flag["overlap"] > 0)
        stop("a subject has overlapping time intervals")
#    if (mcheck$flag["gap"] > 0 || mcheck$flag["jump"] > 0)
#        warning("subject(s) with time gaps, results may be questionable")
```

```
    # The states of the status variable are the first columns in the output
    #  any extra initial states are later in the list.
    # Now that we know the names, verify that p0 is correct (if present)
    if (!missing(p0) && !is.null(p0)) {
        if (length(p0) != nstate) stop("wrong length for p0")
        if (!is.numeric(p0) || abs(1-sum(p0)) > sqrt(.Machine$double.eps))
            stop("p0 must be a numeric vector that adds to 1")
    } else p0 <- NULL
```

The status vector will have values of 0 for censored.

```
⟨survfitCI⟩=
    curves <- vector("list", ncurve)
    names(curves) <- levels(X)

    if (ncol(Y)==2) {  # 1 transition per subject
        # dummy entry time that is < any event time
        t0 <- min(0, Y[,1])
        entry <- rep(t0-1, nrow(Y))
        for (i in levels(X)) {
            indx <- which(X==i)
            curves[[i]] <- docurve2(entry[indx], Y[indx,1], status[indx],
                                    istate[indx], weights[indx],
                                    states,
                                    id[indx], se.fit, influence, p0)
         }
    }
    else {
        ⟨survfitCI-extracens⟩
        ⟨survfitCI-startstop⟩
    }

    ⟨survfitCI-finish⟩
 }
```

In the multi-state case we can calculate the current P(state) vector $p(t)$ using the product-limit form, while the cumulative hazard $c(t)$ is a sum.

$$\begin{aligned}
p(t) &= p(0) \prod_{s<=t} [I + dA(s)] \\
&= p(0) \prod_{s<=t} H(s) \\
c(t) &= \sum_{s<=t} dA(s)
\end{aligned}$$

Where p is a row vector and H is the multi-state hazard matrix. $H(t)$ is a simple transition matrix. Row j of H describes the outcome of everyone who was in state j at time $t-0$; and is the fraction of them who are in states $1, 2, \ldots$ at time $t + 0$. Let $Y_{ij}(t)$ be the indicator function which is 1 if subject i is in state j at time $t - 0$, then

$$H_{jk}(t) = \frac{\sum_i w_i Y_{ij}(t) Y_{ik}(t+)}{\sum_i w_i Y_{ij}(t)} \tag{27}$$

Each row of H sums to 1: everyone has to go somewhere. This formula collapses to the Kaplan-Meier in the simple case where $p(t)$ is a vector of length 2 with state 1 = alive and state 2 = dead.

The variance is based on per-subject influence. Since $p(t)$ is a vector the influence can be written as a matrix with one row per subject and one column per state.

$$U_{ij}(t) \equiv \frac{\partial p_j(t)}{\partial w_i}.$$

This can be calculate using a recursive formula. First, the derivative of a matrix product AB is $d(A)B + Ad(B)$ where $d(A)$ is the elementwise derivative of A and similarly for B. (Write out each element of the matrix product.) Since $p(t) = p(t-)H(t)$, the ith row of U satisfies

$$\begin{aligned} U_i(t) &= \frac{\partial p(t)}{\partial w_i} \\ &= \frac{\partial p(t-)}{\partial w_i} H(t) + p(t-) \frac{\partial H(t)}{\partial w_i} \\ &= U_i(t-) H(t) + p(t-) \frac{\partial H(t)}{\partial w_i} \end{aligned} \tag{28}$$

The first term of 28 collapses to ordinary matrix multiplication. The second term does not: each at risk subject has a unique matrix derivative ∂H; n vectors of length p can be arranged into a matrix, making the code simple, but n p by p matrices are not so neat. However, note that

1. ∂H is zero for anyone not in the risk set, since their weight does not appear in H.
2. Each subject who is at risk will be in one (and only one) of the states at the event time, their weight only appears in that row of H. Thus for each at risk subject ∂H has only one non-zero row.

Say that the subject enters the given event time in state j and ends it in state k. (For most subjects at most time poinnts $k = j$: if there are 100 at risk at time t and 1 changes state, the other 99 stay put.) Let $n_j(t) = \sum_i Y_{ij}(t) w_i$ be the weighted number of subjects in state j, these are the contributers to row j of H. Using equation 27, the derivative of row j with respect to the subject is $(1_k - H_j)/n_j$ where 1_k is a vector with 1 in position k. The product of $p(t)$ with this matrix is the vector $p_j(t)(1_k - H_j)/n_j$. The second term thus turns out to be fairly simple to compute, but I have not seen a way to write it in a compact matrix form

The weighted sum of each column of U will be zero (if computed correctly) and the weighted sum of squares for each column will be the infinitesimal jackknife estimate of variance for the elements of p. The entire variance-covariance matrix for the states is $U'W^2U$ where W is a diagonal matrix of weights, but we currently don't report that back. Note that this is for sampling weights. If one has real case weights, where an integer weight of 2 means 2 observations that were collapsed in to one row of data to save space, then the variance is $U'WU$. Case weights were somewhat common in my youth due to small computer memory, but I haven't seen such data in 20 years.

The residuals for the cumulative hazard are an easier computation, since each hazard function stands alone. In a multistate model with k states there are potentially $k(k-1)$ hazard functions arranged in a k by k matrix, i.e., as used for the NA update; in the code both the hazard, the IJ scores and the standard errors are kept as matrices with a column for each combination that does occur. At each event time only the rows of U2 that correspond to the risk set will be updated.

Below is the function for a single curve. For the status variable a value if 0 is "no event". One nuisance in the function is that we need to ensure the tapply command gives totals for all states, not just the ones present in the data — a call using the `subset` argument might not have all the states — which leads to using factor commands. Another more confusing one is for multiple rows per subject data, where the cstate and U objects have only one row per subject; any given subject is only in one state at a time. This leads to indices of `atrisk` for the set of rows in the risk set but `aindx` for the subjects in the risk set, `death` for the rows that have an event this time and `dindx` for the corresponding subjects.

The setup for (start, stop] data is a bit more work. We want to ensure that a given subject remains in the same group and that they have a continuous period of observation.

If the input data was the result of a tmerge call, say, it might have a lot of extra 'censored' rows. For instance a subject whose state pattern is (0, 5, 1), (5,10, 2), i.e., a transition to state 1 at day 5 and state 2 on day 10 might input as (0,2,0), (2,5,1), (5,6,0), (6,8,0), (8,10,2) instead. These extra censors cause an unnecessary row of output on days 2, 6, and 8. Remove these before going further.

⟨*survfitCI-extracens*⟩=

```
# extra censors
indx <- order(id, Y[,2])   # in stop order
extra <- (survflag(Y[indx,], id[indx]) ==0 & (Y[indx,3] ==0))
# If a subject had obs of (a, b)(b,c)(c,d), and c was a censoring
#  time, that is an "extra" censoring/entry at c that we don't want
#  to count.  Deal with it by changing that subject
#  to (a,b)(b,d).  Won't change S(t), only the n.censored/n.enter count.
if (any(extra)) {
    e2 <- indx[extra]
    Y <- cbind(Y[-(1+e2),1], Y[-e2,-1])
    status <- status[-e2]
    X <- X[-e2]
    id <- id[-e2]
    istate <- istate[-e2]
    weights <- weights[-e2]
    indx <- order(id, Y[,2])
}
```

⟨*survfitCI-startstop*⟩=

```
# Now to work
for (i in levels(X)) {
    indx <- which(X==i)
#    temp <- docurve1(Y[indx,1], Y[indx,2], status[indx],
#                          istate[indx], weights[indx], states, id[indx])
    curves[[i]] <- docurve2(Y[indx,1], Y[indx,2], status[indx],
                            istate[indx],
                            weights[indx], states, id[indx], se.fit,
                            influence, p0)
}
```

⟨*survfitCI-finish*⟩=

```
# Turn the result into a survfit type object
grabit <- function(clist, element) {
    temp <-(clist[[1]][[element]])
    if (is.matrix(temp)) {
        do.call("rbind", lapply(clist, function(x) x[[element]]))
        }
    else {
        xx <- as.vector(unlist(lapply(clist, function(x) x[element])))
        if (inherits(temp, "table")) matrix(xx, byrow=T, ncol=length(temp))
        else xx
    }
}

# we want to rearrange the cumulative hazard to be in time order
#   with one column for each observed transtion.
nstate <- length(states)
temp <- matrix(0, nstate, nstate)
indx1 <- match(rownames(mcheck$transitions), states)
indx2 <- match(colnames(mcheck$transitions), states, nomatch=0) #ignore censor
temp[indx1, indx2[indx2>0]] <- mcheck$transitions[,indx2>0]
ckeep <- which(temp>0)
names(ckeep) <- outer(1:nstate, 1:nstate, paste, sep=’.’)[ckeep]
#browser()

if (length(curves) ==1) {
    keep <- c("n", "time", "n.risk", "n.event", "n.censor", "pstate",
              "p0", "cumhaz", "influence.pstate")
    if (se.fit) keep <- c(keep, "std.err", "sp0")
    kfit <- (curves[[1]])[match(keep, names(curves[[1]]), nomatch=0)]
    names(kfit$p0) <- states
    if (se.fit) kfit$logse <- FALSE
    kfit$cumhaz <- t(kfit$cumhaz[ckeep,,drop=FALSE])
    colnames(kfit$cumhaz) <- names(ckeep)
}
else {
    kfit <- list(n =      as.vector(table(X)),  #give it labels
                 time =   grabit(curves, "time"),
                 n.risk=  grabit(curves, "n.risk"),
                 n.event= grabit(curves, "n.event"),
                 n.censor=grabit(curves, "n.censor"),
                 pstate = grabit(curves, "pstate"),
                 p0     = grabit(curves, "p0"),
                 strata= unlist(lapply(curves, function(x)
                     if (is.null(x$time)) 0L else length(x$time))))
    kfit$p0 <- matrix(kfit$p0, ncol=nstate, byrow=TRUE,
                      dimnames=list(names(curves), states))
    if (se.fit) {
        kfit$std.err <- grabit(curves, "std.err")
        kfit$sp0<- matrix(grabit(curves, "sp0"),
                          ncol=nstate, byrow=TRUE)
        kfit$logse <- FALSE
    }

    # rearrange the cumulative hazard to be in time order, with columns
    #  for each transition
    kfit$cumhaz <- do.call(rbind, lapply(curves, function(x)
        t(x$cumhaz[ckeep,,drop=FALSE])))
    colnames(kfit$cumhaz) <- names(ckeep)

    if (influence) kfit$influence.pstate <-
        lapply(curves, function(x) x$influence.pstate)
}

if (!missing(start.time)) kfit$start.time <- start.time
kfit$transitions <- mcheck$transitions
```

⟨*survfitCI-finish*⟩=

```
#
# Last bit: add in the confidence bands:
#
if (se.fit && conf.type != "none") {
    ci <- survfit_confint(kfit$pstate, kfit$std.err, logse=FALSE,
                            conf.type, conf.int)
    kfit <- c(kfit, ci, conf.type=conf.type, conf.int=conf.int)
}
kfit$states <- states
kfit$type   <- attr(Y, "type")
kfit
```

The updated docurve function is here. One issue that was not recognized originally is delayed entry. If most of the subjects start at time 0, say, but one of them starts at day 100 then that last subject is not a part of p_0. We will define p_0 as the distribution of states just before the first event. The code above has already ensured that each subject has a unique value for istate, so we don't have to search for the right one. The initial vector and leverage are

$$p_0 = (\sum Is_i = 1w_i, \sum Is_i = 2w_i, \ldots)/\sum w_i$$
$$\frac{\partial p_0}{\partial w_k} = [(Is_k = 1, Is_k = 2, ...) - p_0]/\sum w_i$$

The input data set is not necessarily sorted by time or subject. The data has been checked so that subjects don't have gaps, however. The cstate variable for each subject contains their first istate value. Only those intervals that overlap the first event time contribute to p_0. Now: what to report as the "time" for the initial row. The values for it come from (first event time -0), i.e. all who are at risk at the smallest `etime` with status > 0. But for normal plotting the smallest start time seems to be a good default. In the usual (start, stop] data a large chunk of the subjects have a common start time. However, if the first event doesn't happen for a while and subjects are dribbling in, then the best point to start a plot is open to debate. Que sera sera.

⟨*survfitCI-compute*⟩=
```
 docurve2 <- function(entry, etime, status, istate, wt, states, id,
                      se.fit, influence=FALSE, p0) {
     timeset <- sort(unique(etime))
     nstate <- length(states)
     uid <- sort(unique(id))
     index <- match(id, uid)
     # Either/both of id and cstate might be factors.  Data may not be in
     #  order.  Get the initial state for each subject
     temp1 <- order(id, entry)
     temp2 <- match(uid, id[temp1])
     cstate <- (as.numeric(istate)[temp1])[temp2]  # initial state for each

     # The influence matrix can be huge, make sure we have enough memory
     if (influence) {
         needed <- max(nstate * length(uid), 1 + length(timeset))
         if (needed > .Machine$integer.max)
             stop("number of rows for the influence matrix is > the maximum integer")
     }
     storage.mode(wt) <- "double" # just in case someone had integer weights

     # Compute p0 (unless given by the user), and initial leverage matrix
     nid <- length(uid)
     i0  <- matrix(0., nid, nstate)
     if (all(status==0))  t0 <- max(etime)  #failsafe
     else t0 <- min(etime[status!=0])  # first transition event
     at.zero <- (entry < t0 & etime >= t0)
     if (is.null(p0)) {
```

```
    wtsum <- sum(wt[at.zero])  # weights for a subject may change
    p0 <- tapply(wt[at.zero], istate[at.zero], sum) / wtsum
    p0 <- ifelse(is.na(p0), 0, p0)  #for a state not in at.zero, tapply =NA
    if (influence & all(p0 <1)) {  # compute intitial leverage
        who <- index[at.zero]  # this will have no duplicates
        for (j in 1:nstate)
           i0[who,j] <- (ifelse(istate[at.zero]==states[j], 1, 0) - p0[j])/wtsum
    }
}

storage.mode(cstate) <- "integer"
storage.mode(status) <- "integer"
# C code has 0 based subscripts
if (influence) se.fit <- TRUE   # se.fit is free in this case

fit <- .Call(Csurvfitci, c(entry, etime),
             order(entry) - 1L,
             order(etime) - 1L,
             length(timeset),
             status,
             as.integer(cstate) - 1L,
             wt,
             index -1L,
             p0, i0,
             as.integer(se.fit) + 2L*as.integer(influence))

if (se.fit)
    out <- list(n=length(etime), time= timeset, p0 = p0,
                sp0= sqrt(colSums(i0^2)),
         pstate = fit$p, std.err=fit$std,
         n.risk = fit$nrisk,
         n.event= fit$nevent,
         n.censor=fit$ncensor,
         cumhaz = fit$cumhaz)
else out <- list(n=length(etime), time= timeset, p0=p0,
         pstate = fit$p,
         n.risk = fit$nrisk,
         n.event = fit$nevent,
         n.censor= fit$ncensor,
         cumhaz= fit$cumhaz)
if (influence) {
    temp <-  array(fit$influence,
                   dim=c(length(uid), nstate, 1+ length(timeset)),
                   dimnames=list(uid, NULL, NULL))
    out$influence.pstate <- aperm(temp, c(1,3,2))
}
```

```
    out
}
```

9.3.1 C-code

(This is set up as a separate file in the source code directory since it is easier to make emacs stay in C-mode if the file has a .nw extension.)

```
⟨survfitci⟩=
#include "survS.h"
#include "survproto.h"
#include <math.h>

SEXP survfitci(SEXP ftime2,  SEXP sort12,  SEXP sort22, SEXP ntime2,
                    SEXP status2, SEXP cstate2, SEXP wt2,  SEXP id2,
                    SEXP p2,      SEXP i02,     SEXP sefit2) {
    ⟨survfitci-declare⟩
    ⟨survfitci-compute⟩
    ⟨survfitci-return⟩
}
```

Arguments to the routine are the following. For an R object "zed" I use the convention of `zed2` to refer to the object and `zed` to the contents of the object.

ftime A two column matrix containing the entry and exit times for each subject.

sort1 Order vector for the entry times. The first element of sort1 points to the first entry time, etc.

sort2 Order vector for the event times.

ntime Number of unique event time values. This fixes the size of the output arrays.

status Status for each observation. 0= censored

cstate The initial state for each subject, which will be updated during computation to always be the current state.

wt Case weight for each observation.

id The subject id for each observation.

p The initial distribution of states. This will be updated during computation to be the current distribution.

i0 The initial influence matrix, number of subjects by number of states

sefit If 1 then do the se compuatation, if 2 also return the full influence matrix upon which it is based, if 0 the se is not needed.

Note that code is called with id and not cluster: there is a basic premise that each id is a single subject and thus has a unique ”current state” at any given time point. The history of this is that before the survcheck routine, we did not have a good way for a user to normalize the ’current state’ variable for a subject, so this routine takes care of that tracking process. When multi-state Cox models were added we became more formal about this, and users can now have data sets with quite odd patterns of transitions and current state, ones that survcheck calls a teleport. At some point this routine should be updated as well. Cumulative hazard estimates make at least some sense when a subject has a hole, though P(state —t) curves do not.

Declare all of the variables.

⟨*survfitci-declare*⟩=
```
 int i, j, k, kk;   /* generic loop indices */
 int ck, itime, eptr; /*specific indices */
 double ctime;      /*current time of interest, in the main loop */
 int oldstate, newstate; /*when changing state */

 double temp, *temp2;  /* scratch double, and vector of length nstate */
 double *dptr;      /* reused in multiple contexts */
 double *p;         /* current prevalence vector */
 double **hmat;      /* hazard matrix at this time point */
 double **umat=0;     /* per subject leverage at this time point */
 int *atrisk;       /* 1 if the subject is currently at risk */
 int   *ns;         /* number curently in each state */
 int   *nev;        /* number of events at this time, by state */
 double *ws;        /* weighted count of number state */
 double *wtp;       /* case weights indexed by subject */
 double wevent;     /* weighted number of events at current time */
 int nstate;        /* number of states */
 int n, nperson;    /*number of obs, subjects*/
 double **chaz;     /* cumulative hazard matrix */

 /* pointers to the R variables */
 int *sort1, *sort2;  /*sort index for entry time, event time */
 double *entry,* etime;  /*entry time, event time */
 int ntime;          /* number of unique event time values */
 int *status;        /*0=censored, 1,2,... new states */
 int *cstate;        /* current state for each subject */
 int *dstate;        /* the next state, =cstate if not an event time */
 double *wt;         /* weight for each observation */
 double *i0;         /* initial influence */
 int *id;            /* for each obs, which subject is it */
 int sefit;

 /* returned objects */
 SEXP rlist;         /* the returned list and variable names of same */
 const char *rnames[]= {"nrisk","nevent","ncensor", "p",
                        "cumhaz", "std", "influence.pstate", ""};
 SEXP setemp;
 double **pmat, **vmat=0, *cumhaz, *usave=0; /* =0 to silence -Wall warning */
 int  *ncensor, **nrisk, **nevent;
```

Now set up pointers for all of the R objects sent to us. The two that will be updated need to be replaced by duplicates.

```
⟨survfitci-declare⟩=
ntime= asInteger(ntime2);
nperson = LENGTH(cstate2); /* number of unique subjects */
n   = LENGTH(sort12);    /* number of observations in the data */
PROTECT(cstate2 = duplicate(cstate2));
cstate  = INTEGER(cstate2);
entry= REAL(ftime2);
etime= entry + n;
sort1= INTEGER(sort12);
sort2= INTEGER(sort22);
status= INTEGER(status2);
wt = REAL(wt2);
id = INTEGER(id2);
PROTECT(p2 = duplicate(p2));  /*copy of initial prevalence */
p = REAL(p2);
nstate = LENGTH(p2);  /* number of states */
i0 = REAL(i02);
sefit = asInteger(sefit2);

/* allocate space for the output objects
** Ones that are put into a list do not need to be protected
*/
PROTECT(rlist=mkNamed(VECSXP, rnames));
setemp = SET_VECTOR_ELT(rlist, 0, allocMatrix(INTSXP, ntime, nstate));
nrisk =  imatrix(INTEGER(setemp), ntime, nstate);  /* time by state */
setemp = SET_VECTOR_ELT(rlist, 1, allocMatrix(INTSXP, ntime, nstate));
nevent = imatrix(INTEGER(setemp), ntime, nstate);  /* time by state */
setemp = SET_VECTOR_ELT(rlist, 2, allocVector(INTSXP, ntime));
ncensor = INTEGER(setemp);  /* total at each time */
setemp  = SET_VECTOR_ELT(rlist, 3, allocMatrix(REALSXP, ntime, nstate));
pmat =   dmatrix(REAL(setemp), ntime, nstate);
setemp = SET_VECTOR_ELT(rlist, 4, allocMatrix(REALSXP, nstate*nstate, ntime));
cumhaz = REAL(setemp);

if (sefit >0) {
    setemp = SET_VECTOR_ELT(rlist, 5,  allocMatrix(REALSXP, ntime, nstate));
    vmat= dmatrix(REAL(setemp), ntime, nstate);
}
if (sefit >1) {
    /* the max space is larger for a matrix than a vector
    **  This is pure sneakiness: if I allocate a vector then n*nstate*(ntime+1)
    **  may overflow, as it is an integer argument.  Using the rows and cols of
    **  a matrix neither overflows.  But once allocated, I can treat setemp
    **  like a vector since usave is a pointer to double, which is bigger than
    **  integer and won't overflow. */
    setemp = SET_VECTOR_ELT(rlist, 6, allocMatrix(REALSXP, n*nstate, ntime+1));
    usave = REAL(setemp);
}

/* allocate space for scratch vectors */
ws = (double *) R_alloc(2*nstate, sizeof(double)); /*weighted number in state */
temp2 = ws + nstate;
ns    = (int *) R_alloc(2*nstate, sizeof(int));
nev   = ns + nstate;
atrisk = (int *) R_alloc(2*nperson, sizeof(int));
dstate = atrisk + nperson;
wtp = (double *) R_alloc(nperson, sizeof(double));
hmat = (double**) dmatrix((double *)R_alloc(nstate*nstate, sizeof(double)),
                           nstate, nstate);
chaz = (double**) dmatrix((double *)R_alloc(nstate*nstate, sizeof(double)),
                           nstate, nstate);
if (sefit >0)
    umat = (double**) dmatrix((double *)R_alloc(nperson*nstate, sizeof(double)),
                           nstate, nperson);

/* R_alloc does not zero allocated memory */
for (i=0; i<nstate; i++) {
    ws[i] =0;
    ns[i] =0;
    nev[i] =0;
    for (j=0; j<nstate; j++) {
            hmat[i][j] =0;
            chaz[i][j] =0;
    }
}
for (i=0; i<nperson; i++) {
    atrisk[i] =0;
    wtp[i] = 0.0;
    dstate[i] = cstate[i];  /* cstate starts as the initial state */
}
```

Copy over the initial influence data, which was computed in R.

⟨*survfitci-declare*⟩=

```
if (sefit ==1) {
    dptr = i0;
```

```
     for (j=0; j<nstate; j++) {
         for (i=0; i<nperson; i++) umat[i][j] = *dptr++;
     }
  }
  else if (sefit>1) {
      /* copy influence, and save it */
      dptr = i0;
      for (j=0; j<nstate; j++) {
          for (i=0; i<nperson; i++) {
              umat[i][j] = *dptr;
              *usave++ = *dptr++;   /* save in the output */
          }
      }
 }
```

The primary loop of the program walks along the `sort2` vector, with one pass through the interior of the for loop for each unique event time. Observations are at risk in the interval (entry, event]: note the round and square brackets, so a row must satisfy `entry < ctime <= event` to be at risk, where `ctime` is the unique event time of current interest. The basic loop is to add new subjects to the risk set, compute, save results, then remove expired ones from the risk set. The `ns` and `ws` vectors keep track of the number of subjects currently in each state and the weighted number currently in each state. There are four indexing patterns in play which may be confusing.

- The output matrices, indexed by unique event time `itime` and state.
- The `n` observations (variables entry, event, sort1, sort2, status, wt, id)
- The `nperson` individual subjects (variables cstate, atrisk)
- The `[nstate` states (variables hmat, p)

In the code below `i` steps through the exit times and `eptr` the entry time. The `atrisk` variable keeps track of *subjects* who are at risk.

⟨*survfitci-compute*⟩=
```
 itime =0; /*current time index, for output arrays */
 eptr  = 0; /*index to sort1, the entry times */
 for (i=0; i<n; ) {
     ck = sort2[i];
     ctime = etime[ck];  /* current time value of interest */

     /* Add subjects whose entry time is < ctime into the counts */
     for (; eptr<n; eptr++) {
         k = sort1[eptr];
         if (entry[k] < ctime) {
             kk = cstate[id[k]];  /*current state of the addition */
             ns[kk]++;
```

```
            ws[kk] += wt[k];
            wtp[id[k]] = wt[k];
            atrisk[id[k]] =1;   /* mark them as being at risk */
        }
        else break;
    }

    ⟨survfitci-compute-matrices⟩
    ⟨survfitci-compute-update⟩

    /* Take the current events and censors out of the risk set */
    for (; i<n; i++) {
        j= sort2[i];
        if (etime[j] == ctime) {
            oldstate = cstate[id[j]]; /*current state */
            ns[oldstate]--;
            ws[oldstate] -= wt[j];
            if (status[j] >0) cstate[id[j]] = status[j]-1; /*new state */
            atrisk[id[j]] =0;
        }
        else break;
    }
    itime++;
}
```

The key variables for the computation are the matrix H and the current prevalence vector P. H is created anew at each unique time point. Row j of H concerns everyone in state j just before the time point, and contains the transitions at that time point. So the jk element is the (weighted) fraction who change from state j to state k, and the jj element the fraction who stay put. Each row of H by definition sums to 1. If no one is in the state then the jj element is set to 1. A second version which we call H2 has 1 subtracted from each diagonal giving row sums are 0, we go back and forth depending on which is needed at the moment. If there are no events at this time point P and U do not update.

⟨*survfitci-compute-matrices*⟩=

```
for (j=0; j<nstate; j++) {
    for (k=0; k<nstate; k++) {
        hmat[j][k] =0;
    }
 }

/* Count up the number of events and censored at this time point */
for (k=0; k<nstate; k++) nev[k] =0;
ncensor[itime] =0;
wevent =0;
for (j=i; j<n; j++) {
    k = sort2[j];
    if (etime[k] == ctime) {
        if (status[k] >0) {
            newstate = status[k] -1;  /* 0 based subscripts */
            oldstate = cstate[id[k]];
            if (oldstate != newstate) {
                /* A "move" to the same state does not count */
                dstate[id[k]] = newstate;
                nev[newstate]++;
                wevent += wt[k];
                hmat[oldstate][newstate] += wt[k];
            }
        }
        else ncensor[itime]++;
    }
    else break;
 }

if (wevent > 0) {  /* there was at least one move with weight > 0 */
    /* finish computing H */
    for (j=0; j<nstate; j++) {
        if (ns[j] >0) {
            temp =0;
            for (k=0; k<nstate; k++) {
                temp += hmat[j][k];
                hmat[j][k] /= ws[j];  /* events/n */
            }
            hmat[j][j] =1 -temp/ws[j]; /*rows sum to one */
        }
        else hmat[j][j] =1.0;

    }
    if (sefit >0) {
        ⟨survfitci-compute-U⟩
    }
    ⟨survfitci-compute-P⟩
}
```

The most complicated part of the code is the update of the per subject influence matrix U. The influence for a subject is the derivative of the current estimates wrt the case weight of that subject. Since p is a vector the influence U is easily represented as a matrix with one row per subject and one column per state. Refer to equation (28) for the derivation.

Let m and n be the old and new states for subject i, and n_m the sum of weights for all subjects at risk in state m. Then

$$U_{ij}(t) = \sum_k [U_{ik}(t-)H_{kj}] + p_m(t-)(I_{n=j} - H_{mj})/n_m$$

1. The first term above is simple matrix multiplication.
2. The second adds a vector with mean zero.

If standard errors are not needed we can skip this calculation.

⟨*survfitci-compute-U*⟩=
```
/* Update U, part 1  U = U %*% H -- matrix multiplication */
for (j=0; j<nperson; j++) { /* row of U */
        for (k=0; k<nstate; k++) { /* column of U */
            temp2[k]=0;
            for (kk=0; kk<nstate; kk++)
                temp2[k] += umat[j][kk] * hmat[kk][k];
        }
        for (k=0; k<nstate; k++) umat[j][k] = temp2[k];
}

/* step 2, add in dH term */
for (j=0; j<nperson; j++) {
        if (atrisk[j]==1) {
        oldstate = cstate[j];
            for (k=0; k<nstate; k++)
                umat[j][k] -= hmat[oldstate][k]* p[oldstate]/ ws[oldstate];
            umat[j][dstate[j]] += p[oldstate]/ws[oldstate];
        }
}
```

Now update the cumulative hazard by adding H2 to it, and update p to pH.

⟨*survfitci-compute-P*⟩=
```
/* Finally, update chaz and p.  */
for (j=0; j<nstate; j++) {
    for (k=0; k<nstate; k++) chaz[j][k] += hmat[j][k];
    chaz[j][j] -=1;  /* Update using H2 */

    temp2[j] =0;
    for (k=0; k<nstate; k++)
        temp2[j] += p[k] * hmat[k][j];
 }
for (j=0; j<nstate; j++) p[j] = temp2[j];
```

⟨*survfitci-compute-update*⟩=
```
/* store into the matrices that will be passed back */
for (j=0; j<nstate; j++) {
    pmat[j][itime] = p[j];
    nrisk[j][itime] = ns[j];
    nevent[j][itime] = nev[j];
    for (k=0; k<nstate; k++) *cumhaz++ = chaz[k][j];
```

```
     if (sefit >0) {
         temp =0;
         for (k=0; k<nperson; k++)
             temp += wtp[k]* wtp[k]*umat[k][j]*umat[k][j];
         vmat[j][itime] = sqrt(temp);
     }
     if (sefit > 1)
         for (k=0; k<nperson; k++) *usave++ = umat[k][j];
  }
```

```
⟨survfitci-return⟩=
 /* return a list */
 UNPROTECT(3);
 return(rlist);
```

9.3.2 Printing and plotting

The survfitms class differs from a survfit, but many of the same methods nearly apply.

```
⟨survfitms⟩=
 # Methods for survfitms objects
 ⟨survfitms-summary⟩
 ⟨survfitms-subscript⟩
```

The subscript method is a near copy of that for survfit objects, but with a slightly different set of components. The object could have strata and will almost always have multiple columns. Following convention, if there is only one subscript we treat the object as though it were a vector. The nmatch function allow the user to use either names or integer indices.

```
⟨survfitms-subscript⟩=
 "[.survfitms" <- function(x, ..., drop=FALSE) {
     nmatch <- function(i, target) {
         # This function lets R worry about character, negative,
         # or logical subscripts
         #  It always returns a set of positive integer indices
         temp <- seq(along.with=target)
         names(temp) <- target
         temp[i]
     }
     if (!is.null(x$influence.pstate) || !is.null(x$influence.cumhaz))
         x <- survfit0(x, x$start.time)  # make influence and pstate align
     ndots <- ...length()      # the simplest, but not avail in R 3.4
     # ndots <- length(list(...))# fails if any are missing, e.g. fit[,2]
     # ndots <- if (missing(drop)) nargs()-1 else nargs()-2  # a workaround

     dd <- dim(x)
     dmatch <- match(c("strata", "data", "states"), names(dd), nomatch=0)
```

```
if (is.null(x$states)) stop("survfitms object has no states component")
if (dmatch[3]==0) stop ("survfitms object has no states dimension")
dtype <- match(names(dd), c("strata", "data", "states"))

if (ndots==0) return(x)  # no subscript given
if (ndots >0 && !missing(..1)) i <- ..1 else i <- NULL
if (ndots> 1 && !missing(..2)) j <- ..2 else j <- NULL
if (ndots> 2 && !missing(..3)) k <- ..3 else k <- NULL
if (is.null(i) & is.null(j) & is.null(k)) return(x) # only one curve

# Make a new object
newx <- vector("list", length(x))
names(newx) <- names(x)
for (kk in c("logse", "version", "conf.int", "conf.type", "type",
             "start.time", "call"))
    if (!is.null(x[[kk]])) newx[[kk]] <- x[[kk]]
newx$transitions <- NULL # may no longer be accurate, and not needed
class(newx) <- class(x)

# Like a matrix, let the user use a single subscript if they desire
if (ndots==1 && length(dd) > 1) {
    # the ’treat it as a vector’ case
    if (!is.numeric(i))
        stop("single subscript must be numeric")
    if (any(dmatch==2)) stop("single index subscripts are not supported for a survfit objet wi

    # when subscripting a mix, these don’t endure
    newx$cumhaz <- newx$std.chaz <- newx$influence.chaz <- NULL
    newx$transitions <- newx$states <- newx$newdata <- NULL

    # what strata and columns do I need?
    itemp <- matrix(1:prod(dd), nrow=dd[1])
    jj <- (col(itemp))[i]    # columns
    ii <- (row(itemp))[i]    # this is now the strata id

    if (dtype[1]!=1 || dd[1]==1) # no strata or only 1
        irow <- rep(seq(along.with= x$time), length(ii))
    else {
        itemp2 <- split(1:sum(x$strata), rep(1:length(x$strata), x$strata))
        irow <- unlist(itemp2[ii])  # rows of the pstate object
    }
    inum <- x$strata[ii]        # number of rows in each ii
    indx <- cbind(irow, rep(jj,ii))      # matrix index for pstate

    # The n.risk, n.event, .. matrices dont have a newdata dimension.
    if (all(dtype!=2) || dd["data"]==1) kk <- jj
    else {  # both data and states
        itemp <- matrix(1:(dd["data"]*dd["states"]), nrow=dd[2])
        kk <- (col(itemp))[jj]    # the state of each selected one
        indx2 <- cbind(irow, rep(k, irow))
    }
    newx$n <- x$n[ii]
    newx$time <- x$time[irow]
    for (z in c("n.risk", "n.event", "n.censor", "n.enter"))
        if (!is.null(x[[z]])) newx[[z]] <- (x[[z]])[indx2]
    for (z in c("pstate", "std.err", "upper", "lower"))
        if (!is.null(x[[z]])) newx[[z]] <- (x[[z]])[indx]

    newx$strata <- x$strata[ii]
    names(newx$strata) <- seq(along.with=ii)

    return(newx)
}

# not a single subscript, i.e., the usual case
# Backwards compatability: If x$strata=NULL, it is a semantic argument
#  of whether there is still "1 stratum".  I have used the second
#  form at times, e.g.  x[1,,2] for an object with only data and state
#  dimensions.
# If there are no strata, 1 too many subscripts, and the first is 1,
#  assume this case and toss the first
if (ndots == (length(dd)+1)) {
    if (is.null(x$strata) && (is.null(i) || (length(i)==1 && i==1))) {
        i <-j; j <-k; k <- NULL
    } else stop("incorrect number of dimensions")
} else if (ndots != length(dd)) stop("incorrect number of dimensions")

# create irow, which selects for the time dimension of x
if (dtype[1]!=1 || is.null(i)) {
    irow <- seq(along.with= x$time)
}
else {
    i <- nmatch(i, names(x$strata))
    itemp <- split(1:sum(x$strata), rep(1:length(x$strata), x$strata))
    irow <- unlist(itemp[i])  # rows of the pstate object
}

# Select the n, strata, and time components of the output.  Make j,k
#  point to the subscripts other than strata (makes later code a touch
#  simpler.)
newx$time <- x$time[irow]
if (dtype[1] !=1) {  # there are no strata
```

```
    newx$n <- x$n
    k <- j; j <- i;
    dd <- c(0, dd)
    dtype <- c(1, dtype)
}
else { # there are strata
    if (is.null(i)) i <-seq(along.with=x$strata)
    if ((drop && length(i)>1) || !drop) newx$strata <- x$strata[i]
    newx$n <- x$n[i]
}

# The n.censor and n.enter values do not repeat with multiple X values
for (z in c("n.censor", "n.enter"))
    if (!is.null(x[[z]])) newx[[z]] <- (x[[z]])[irow, drop=FALSE]

# two cases: with newx or without newx  (pstate is always present)
nstate <- length(x$states)
if (dtype[2] !=2) {  # j indexes the states, there is no data dimension
    if (is.null(j)) j <- seq.int(nstate)
    else j <- nmatch(j, x$states)

    # keep these as start points for plotting, even though they won't make
    #  true sense if states are subset, since rows won't sum to 1
    if (!is.null(x$p0)) {
        if (is.matrix(x$p0)) newx$p0 <- x$p0[i,j, drop=FALSE]
        else newx$p0 <- x$p0[j]
    }
    if (!is.null(x$sp0)) {
        if (is.matrix(x$sp0)) newx$sp0 <- x$sp0[i,j, drop=FALSE]
        else newx$sp0 <- x$sp0[j]
    }

    # in the rare case of a single strata with 1 obs, don't drop dims
    if (length(irow)==1 && length(j) > 1) drop2 <- FALSE
    else drop2 <- drop

    for (z in c("n.risk", "n.event"))
        if (!is.null(x[[z]])) newx[[z]] <- (x[[z]])[irow,j, drop=drop2]
    for (z in c("pstate", "std.err", "upper", "lower"))
        if (!is.null(x[[z]])) newx[[z]] <- (x[[z]])[irow,j, drop=drop2]
    if (!is.null(x$influence.pstate)) {
        if (is.list(x$influence.pstate)) {
            if (length(i)==1) newx$influence.pstate <- x$influence.pstate[[i]]
            else newx$influence.pstate <- lapply(x$influence.pstate[i],
                                    function(x) x[,,j, drop= drop])
            }
        else newx$influence.pstate <- x$influence.pstate[,,j, drop=drop]
    }

    if (length(j)== nstate && all(j == seq.int(nstate))) {
        # user kept all the states, in original order
        newx$states <- x$states
        for (z in c("cumhaz", "std.chaz"))
             if (!is.null(x[[z]])) newx[[z]] <- (x[[z]])[irow,, drop=drop2]
        if (!is.null(x$influence.chaz)) {
            if (is.list(x$influence.chaz)) {
                newx$influence.chaz <- x$influence.chaz[i]
                if (length(i)==1 && drop)
                    newx$influence.chaz <- x$influence.chaz[[i]]
            }
            else newx$influence.chaz <- x$influence.chaz
        }
    }
    else {
        # Some states were dropped, leaving no consistent way to
        #  subscript cumhaz, or not one I have yet seen clearly
        # So remove it from the object
        newx$cumhaz <- newx$std.chaz <- newx$influence.chaz <- NULL
        if (length(j)==1 & drop) {
            newx$states <- NULL
            temp <- class(newx)
            class(newx) <- temp[temp!="survfitms"]
        }
        else newx$states <- x$states[j]
    }
}
else {  # j points at newdata, k points at states
    if (is.null(j)) j <- seq.int(dd[2])
    else j <- nmatch(j, seq.int(dd[2]))

    if (is.null(k)) k <- seq.int(nstate)
    else k <- nmatch(k, x$states)

    # keep these as start points for plotting, even though they won't make
    #  true sense is states are subset, since rows won't sum to 1
    # (all data= sets have the same p0)
    if (!is.null(x$p0)) {
        if (is.matrix(x$p0)) newx$p0 <- x$p0[i,k] else newx$p0 <- x$p0[k]
    }
    if (!is.null(x$sp0)) {
        if (is.matrix(x$sp0)) newx$sp0 <- x$p0[i,k] else newx$sp0 <- x$sp0[k]
    }
```

```
 if (length(irow)==1) {
    if (length(j) > 1) drop2 <- FALSE else drop2<- drop
    if (length(k) > 1) drop3 <- FALSE else drop3 <- drop
}
else drop2 <- drop3 <- drop

for (z in c("n.risk", "n.event"))
    if (!is.null(x[[z]])) newx[[z]] <- (x[[z]])[irow, k, drop=drop3]
for (z in c("pstate", "std.err", "upper", "lower"))
    if (!is.null(x[[z]])) newx[[z]] <- (x[[z]])[irow,j,k, drop=drop2]

if (!is.null(x$influence.pstate)) {
    if (is.list(x$influence.pstate)) {
        if (length(i)==1)
            newx$influence.pstate <- (x$influence.pstate[[i]])[,,j,k, drop=drop]
        else newx$influence.pstate <- lapply(x$influence.pstate[i],
                             function(x) x[,,j,k, drop= drop])
        }
    else newx$influence.pstate <- x$influence.pstate[,,j,k, drop=drop]
}

if (length(k)== nstate && all(k == seq.int(nstate))) {
    # user kept all the states
    newx$states <- x$states
    for (z in c("cumhaz", "std.chaz"))
         if (!is.null(x[[z]]))
             newx[[z]] <- (x[[z]])[irow,j,, drop=drop2]
    if (!is.null(x$influence.chaz)) {
        if (is.list(x$influence.chaz)) {
            newx$influence.chaz <- (x$influence.chaz[i])[,j,]
            if (length(i)==1 && drop)
                newx$influence.chaz <- x$influence.chaz[[i]]
        }
        else newx$influence.chaz <- x$influence.chaz[,j,]
    }
}
else {
    # never drop the states component.  Otherwise downstream code
    #  will start looking for x$surv instead of x$pstate
    newx$states <- x$states[k]
    newx$cumhaz <- newx$std.chaz <- newx$influence.chaz <- NULL
    x$transitions <- NULL
 }

if (length(j)==1 && drop) newx$newdata <- NULL
```

```
        else newx$newdata <- x$newdata[j,,drop=FALSE]  #newdata is a data frame

    }
    newx
}
```

The summary.survfit and summary.survfitms functions share a significant amount of code. One part of the code that once was subtle is dealing with intermediate time points; the findInterval function in base R has made that much easier. Since the result does not involve interpolation, one should be able to create a special index vector i and return `time[i]`, `surv[i,]`, etc, to subscript all the curves in a survfit object at once. But that approach, though efficient in theory, runs into two problems. First is the extrapolated value for the curves at time points before the first event, which is allowed to be different for different curves in survfitms objects. The second is that there is interpolation of a sort: the n.event and n.censor components are summed over intervals when the selected time points are sparse, and that process is very tricky for multiple curves at once. At one point the code took that approach, but it became too complex to maintain. The current approach is slower but more transparent: do the individual curves one by one, then paste together the results.

⟨*survfitms-summary*⟩=
```
summary.survfit <- function(object, times, censored=FALSE,
                            scale=1, extend=FALSE,
                            rmean=getOption('survfit.rmean'),
                            ...) {
    fit <- object  #save typing
    if (!inherits(fit, 'survfit'))
            stop("summary.survfit can only be used for survfit objects")
    if (is.null(fit$logse)) fit$logse <- TRUE   #older style

    # The print.rmean option is depreciated, it is still listened
    #   to in print.survfit, but ignored here
    if (is.null(rmean)) rmean <- "common"
    if (is.numeric(rmean)) {
        if (is.null(fit$start.time)) {
            if (rmean < min(fit$time))
                stop("Truncation point for the mean time in state is < smallest survival")
        }
        else if (rmean < fit$start.time)
            stop("Truncation point for the mean time in state is < smallest survival")
    }
    else {
        rmean <- match.arg(rmean, c('none', 'common', 'individual'))
        if (length(rmean)==0) stop("Invalid value for rmean option")
    }

    # adding time 0 makes the mean and median easier
```

```
fit0 <- survfit0(fit, fit$start.time)  #add time 0
temp <- survmean(fit0, scale=scale, rmean)
table <- temp$matrix  #for inclusion in the output list
rmean.endtime <- temp$end.time

if (!is.null(fit$strata)) {
    nstrat <-  length(fit$strata)
}
delta <- function(x, indx) {  # sums between chosen times
    if (is.logical(indx)) indx <- which(indx)
    if (!is.null(x) && length(indx) >0) {
        fx <- function(x, indx) diff(c(0, c(0, cumsum(x))[indx+1]))
        if (is.matrix(x)) {
            temp <- apply(x, 2, fx, indx=indx)
            # don't return a vector when only 1 time point is given
            if (is.matrix(temp)) temp else matrix(temp, nrow=1)
        }
        else fx(x, indx)
    }
    else NULL
}

if (missing(times)) {
    ⟨survfitms-simple⟩
}
else {
    fit <- fit0
    ⟨survfitms-times⟩
    times <- sort(times)  #in case the user forgot
    if (is.null(fit$strata)) fit <- findrow(fit, times, extend)
    else {
        ltemp <- vector("list", nstrat)
        for (i in 1:nstrat)
            ltemp[[i]] <- findrow(fit[i], times, extend)
        fit <- unpacksurv(fit, ltemp)
    }
}

# finish off the output structure
fit$table <- table
if (length(rmean.endtime)>0  && !any(is.na(rmean.endtime[1])))
        fit$rmean.endtime <- rmean.endtime

# A survfit object may contain std(log S) or std(S), summary always std(S)
if (!is.null(fit$std.err) && fit$logse) fit$std.err <- fit$std.err * fit$surv
```

```
    # Expand the strata
    if (!is.null(fit$strata))
        fit$strata <- factor(rep(1:nstrat, fit$strata), 1:nstrat,
                             labels= names(fit$strata))
    if (scale != 1) {
        # fix scale in the output
        fit$time <- fit$time/scale
    }

    class(fit) <- "summary.survfit"
    fit
}
```

The simple case of no times argument.

⟨*survfitms-simple*⟩=

```
if (!censored) {
    index <- (rowSums(as.matrix(fit$n.event)) >0)
    for (i in c("time","n.risk", "n.event", "surv", "pstate", "std.err",
                    "upper", "lower", "cumhaz", "std.chaz")) {
        if (!is.null(fit[[i]])) {  # not all components in all objects
            temp <- fit[[i]]
            if (is.matrix(temp)) temp <- temp[index,,drop=FALSE]
            else  if (!is.array(temp)) temp <- temp[index]  #simple vector
            else temp <- temp[index,,, drop=FALSE] # 3 way
            fit[[i]] <- temp
        }
    }

    # The n.enter and n.censor values are accumualated
    #  both of these are simple vectors
    if (is.null(fit$strata)) {
        for (i in c("n.enter", "n.censor"))
            if (!is.null(fit[[i]]))
                fit[[i]] <- delta(fit[[i]], index)
    }
    else {
        sindx <- rep(1:nstrat, fit$strata)
        for (i in c("n.enter", "n.censor")) {
            if (!is.null(fit[[i]]))
                fit[[i]] <- unlist(sapply(1:nstrat, function(j)
                             delta(fit[[i]][sindx==j], index[sindx==j])))
        }
        # the "factor" is needed for the case that a strata has no
        #  events at all, and hence 0 lines of output
        fit$strata[] <- as.vector(table(factor(sindx[index], 1:nstrat)))
    }
```

```
 }
 #if missing(times) and censored=TRUE, the fit object is ok as it is
```

To deal with selected times we first define a subscripting function. For indices of 0, which are requested times that are before the first event, it fills in the initial value.

⟨*survfitms-times*⟩=
```
 ssub<- function(x, indx) {  #select an object and index
     if (!is.null(x) && length(indx)>0) {
         if (is.matrix(x)) x[pmax(1,indx),,drop=FALSE]
         else if (is.array(x))  x[pmax(1,indx),,,drop=FALSE]
         else x[pmax(1, indx)]
     }
     else NULL
 }
```

This function does the real work, for any single curve. The default value for init is correct for survival curves.

Say that the data has values at time 5, 10, 15, 20 …, and a user asks for `times=c(7, 15, 20, 30)`. In the input object `n.risk` refers to the number at risk just before time 5, 10, …; it is a left-continuous function. The survival is a right-continuous function. So at time 7 we want to take the survival from time 5 and number at risk from time 10; `indx1` will be the right-continuous index and `indx2` the left continuous one. The value of n.risk at time 30 has to be computed. For counts of events, censoring, and entry we want to know the total number that happened during the intervals of 0-7, 7-15, 15-20 and 20-30. Technically censorings at time 15 happen just after time 15 so would go into the third line of the report. However, this would lead to terrible confusion for the user since using `times=c(5, 10, 15, 20)` would lead to different counts than a call that did not contain the times argument, so all 3 of the intermediates are computed using indx1. A report at time 30 is made only if extend=TRUE, in which case we need to compute a tail value for n.risk.

⟨*survfitms-times*⟩=
```
 findrow <- function(fit, times, extend) {
     if (FALSE) {
         if (is.null(fit$start.time)) mintime <- min(fit$time, 0)
         else                         mintime <- fit$start.time
         ptimes <- times[times >= mintime]
     }  else ptimes <- times[is.finite(times)]

     if (!extend) {
         maxtime <- max(fit$time)
         ptimes <- ptimes[ptimes <= maxtime]
     }
     ntime <- length(fit$time)

     index1 <- findInterval(ptimes, fit$time)
     index2 <- 1 + findInterval(ptimes, fit$time, left.open=TRUE)
```

```
    if (length(index1) ==0)
        stop("no points selected for one or more curves, consider using the extend argument")
    # The pmax() above encodes the assumption that n.risk for any
    #  times before the first observation = n.risk at the first obs
    fit$time <- ptimes

    for (i in c("surv", "pstate", "upper", "lower", "std.err", "cumhaz",
                "std.chaz")) {
        if (!is.null(fit[[i]])) fit[[i]] <- ssub(fit[[i]], index1)
    }

    if (is.matrix(fit$n.risk)) {
        # Every observation in the data has to end with a censor or event.
        #  So by definition the number at risk after the last observed time
        #  value must be 0.
        fit$n.risk <- rbind(fit$n.risk,0)[index2,,drop=FALSE]
    }
    else  fit$n.risk <- c(fit$n.risk, 0)[index2]

    for (i in c("n.event", "n.censor", "n.enter"))
        fit[[i]] <- delta(fit[[i]], index1)
    fit
}

# For a single component, turn it from a list into a single vector, matrix
#  or array
unlistsurv <- function(x, name) {
    temp <- lapply(x, function(x) x[[name]])
    if (is.vector(temp[[1]])) unlist(temp)
    else if (is.matrix(temp[[1]])) do.call("rbind", temp)
    else {
        # the cumulative hazard is the only component that is an array
        # it's third dimension is n
        xx <- unlist(temp)
        dd <- dim(temp[[1]])
        dd[3] <- length(xx)/prod(dd[1:2])
        array(xx, dim=dd)
    }
}

# unlist all the components built by a set of calls to findrow
#  and remake the strata
unpacksurv <- function(fit, ltemp) {
    keep <- c("time", "surv", "pstate", "upper", "lower", "std.err",
              "cumhaz", "n.risk", "n.event", "n.censor", "n.enter",
              "std.chaz")
    for (i in keep)
        if (!is.null(fit[[i]])) fit[[i]] <- unlistsurv(ltemp, i)
    fit$strata[] <- sapply(ltemp, function(x) length(x$time))
    fit
}
```

Repeat the code for survfitms objects. The only real difference is the preservation of pstate and cumhaz instead of surv, and the use of survmean2.

⟨*survfitms-summary*⟩=
```
summary.survfitms <- function(object, times, censored=FALSE,
                              scale=1, extend=FALSE,
                              rmean= getOption("survfit.rmean"),
                              ...) {

    fit <- object  # save typing
    if (!inherits(fit, 'survfitms'))
            stop("summary.survfitms can only be used for survfitms objects")
    if (is.null(fit$logse)) fit$logse <- FALSE  # older style

    # The print.rmean option is depreciated, it is still listened
    #   to in print.survfit, but ignored here
    if (is.null(rmean)) rmean <- "common"
    if (is.numeric(rmean)) {
        if (is.null(fit$start.time)) {
            if (rmean < min(fit$time))
                stop("Truncation point for the mean is < smallest survival")
        }
        else if (rmean < fit$start.time)
            stop("Truncation point for the mean is < smallest survival")
    }
    else {
        rmean <- match.arg(rmean, c('none', 'common', 'individual'))
        if (length(rmean)==0) stop("Invalid value for rmean option")
    }

    fit0 <- survfit0(fit, fit$start.time) # add time 0
    temp <- survmean2(fit0, scale=scale, rmean)
    table <- temp$matrix  #for inclusion in the output list
    rmean.endtime <- temp$end.time

    if (!missing(times)) {
        if (!is.numeric(times)) stop ("times must be numeric")
        times <- sort(times)
    }

    if (!is.null(fit$strata)) {
```

```
    nstrat <-  length(fit$strata)
    sindx <- rep(1:nstrat, fit$strata)
}
delta <- function(x, indx) {  # sums between chosen times
    if (is.logical(indx)) indx <- which(indx)
    if (!is.null(x) && length(indx) >0) {
        fx <- function(x, indx) diff(c(0, c(0, cumsum(x))[indx+1]))
        if (is.matrix(x)) {
            temp <- apply(x, 2, fx, indx=indx)
            if (is.matrix(temp)) temp else matrix(temp, nrow=1)
        }
        else fx(x, indx)
    }
    else NULL
}

if (missing(times)) {
    ⟨survfitms-simple⟩
}
else {
    fit <-fit0  # easier to work with
    ⟨survfitms-times⟩
    times <- sort(times)
    if (is.null(fit$strata)) fit <- findrow(fit, times, extend)
    else {
        ltemp <- vector("list", nstrat)
        for (i in 1:nstrat)
            ltemp[[i]] <- findrow(fit[i,], times, extend)
        fit <- unpacksurv(fit, ltemp)
    }
}

# finish off the output structure
fit$table <- table
if (length(rmean.endtime)>0  && !any(is.na(rmean.endtime)))
        fit$rmean.endtime <- rmean.endtime

if (!is.null(fit$strata))
    fit$strata <- factor(rep(names(fit$strata), fit$strata))

# A survfit object may contain std(log S) or std(S), summary always std(S)
if (!is.null(fit$std.err) && fit$logse) fit$std.err <- fit$std.err * fit$surv
if (scale != 1) {
    # fix scale in the output
    fit$time <- fit$time/scale
}
```

```
     class(fit) <- "summary.survfitms"
     fit
 }

 ⟨printms⟩
 ⟨survmean2⟩
```

Printing for a survfitms object is different than for a survfit one. The big difference is that I don't have an estimate of the median, or any other quantile for that matter. Mean time in state makes sense, but I don't have a standard error for it at the moment. The other is that there is usually a mismatch between the n.event matrix and the n.risk matrix. The latter has all the states that were possible whereas the former only has states with an arrow pointing in. We need to manufacture the 0 events for the other states.

```
⟨printms⟩=
 print.survfitms <- function(x, scale=1,
                             rmean = getOption("survfit.rmean"), ...) {
     if (!is.null(cl<- x$call)) {
         cat("Call: ")
         dput(cl)
         cat("\n")
         }
     omit <- x$na.action
     if (length(omit)) cat("  ", naprint(omit), "\n")

     x <- survfit0(x, x$start.time)
     if (is.null(rmean)) rmean <- "common"
     if (is.numeric(rmean)) {
         if (is.null(x$start.time)) {
             if (rmean < min(x$time))
                 stop("Truncation point for the mean is < smallest survival")
         }
         else if (rmean < x$start.time)
             stop("Truncation point for the mean is < smallest survival")
     }
     else {
         rmean <- match.arg(rmean, c('none', 'common', 'individual'))
         if (length(rmean)==0) stop("Invalid value for rmean option")
     }

     temp <- survmean2(x, scale=scale, rmean)
     if (is.null(temp$end.time)) print(temp$matrix, ...)
     else {
         etime <- temp$end.time
         dd <- dimnames(temp$matrix)
         cname <- dd[[2]]
```

```
        cname[length(cname)] <- paste0(cname[length(cname)], '*')
        dd[[2]] <- cname
        dimnames(temp$matrix) <- dd
        print(temp$matrix, ...)
        if (length(etime) ==1)
             cat("   *restricted mean time in state (max time =",
                 format(etime, ...), ")\n")
        else cat("   *restricted mean time in state (per curve cutoff)\n")
    }
    invisible(x)
}
```

This part of the computation is set out separately since it is called by both print and summary.

⟨*survmean2*⟩=
```
survmean2 <- function(x, scale=1, rmean) {
    nstate <- length(x$states)  #there will always be at least 1 state
    ngrp   <- max(1, length(x$strata))
    if (is.null(x$newdata)) ndata <- 0  else ndata <- nrow(x$newdata)
    if (ngrp >1)  {
        igrp <- rep(1:ngrp, x$strata)
        rname <- names(x$strata)
        }
    else {
        igrp <- rep(1, length(x$time))
        rname <- NULL
        }

    # The n.event matrix may not have nstate columms.  Its
    #  colnames are the first elements of states, however
    if (is.matrix(x$n.event)) {
        nc <- ncol(x$n.event)
        nevent <- tapply(x$n.event, list(rep(igrp, nc), col(x$n.event)), sum)
        dimnames(nevent) <- list(rname, x$states[1:nc])
        }
    else {
        nevent <- tapply(x$n.event, igrp, sum)
        names(nevent) <- rname
        }

    if (ndata< 2) {
        outmat <- matrix(0., nrow=nstate*ngrp , ncol=2)
        outmat[,1] <- rep(x$n, nstate)
        outmat[1:length(nevent), 2] <- c(nevent)

        if (ngrp >1)
            rowname <- c(outer(rname, x$states, paste, sep=", "))
        else rowname <- x$states
    }
    else {
        outmat <- matrix(0., nrow=nstate*ndata*ngrp, ncol=2)
        outmat[,1] <- rep(x$n, nstate*ndata)
        outmat[, 2] <- rep(c(nevent), each=ndata)

        temp <- outer(1:ndata, x$states, paste, sep=", ")
        if (ngrp >1)
            rowname <- c(outer(rname, temp, paste, sep=", "))
        else rowname <- temp
        nstate <- nstate * ndata
    }

    # Caculate the mean time in each state
    if (rmean != "none") {
        if (is.numeric(rmean)) maxtime <- rep(rmean, ngrp)
        else if (rmean=="common") maxtime <- rep(max(x$time), ngrp)
        else maxtime <- tapply(x$time, igrp, max)

        meantime <- matrix(0., ngrp, nstate)
        if (!is.null(x$influence)) stdtime <- meantime
        for (i in 1:ngrp) {
            # a 2 dimensional matrix is an "array", but a 3-dim array is
            #  not a "matrix", so check for matrix first.
            if (is.matrix(x$pstate))
                temp <- x$pstate[igrp==i,, drop=FALSE]
            else if (is.array(x$pstate))
                temp <- matrix(x$pstate[igrp==i,,,drop=FALSE],
                               ncol= nstate)
            else temp <- matrix(x$pstate[igrp==i], ncol=1)

            tt <- x$time[igrp==i]

            # Now cut it off at maxtime
            delta <- diff(c(tt[tt<maxtime[i]], maxtime[i]))
            if (length(delta) > nrow(temp)) delta <- delta[1:nrow(temp)]
            if (length(delta) < nrow(temp))
                delta <- c(delta, rep(0, nrow(temp) - length(delta)))
            meantime[i,] <- colSums(delta*temp)

            if (!is.null(x$influence)) {
                # calculate the variance
                if (is.list(x$influence))
                    itemp <- apply(x$influence[[i]], 1,
                                   function(x) colSums(x*delta))
                else itemp <- apply(x$influence, 1,
                                    function(x) colSums(x*delta))
                stdtime[i,] <- sqrt(rowSums(itemp^2))
            }
        }
        outmat <- cbind(outmat, c(meantime)/scale)
        cname <- c("n", "nevent", "rmean")
        if (!is.null(x$influence)) {
            outmat <- cbind(outmat, c(stdtime)/scale)
            cname <- c(cname, "se(rmean)")
        }
        # report back a single time, if there is only one
        if (all(maxtime == maxtime[1])) maxtime <- maxtime[1]
    }
    else cname <- c("n", "nevent")
    dimnames(outmat) <- list(rowname, cname)

    if (rmean=='none') list(matrix=outmat)
    else list(matrix=outmat, end.time=maxtime/scale)
}
```

10 Matrix exponentials and transition matrices

For multi-state models, we need to compute the exponential of the transition matrix, sometimes many times. The matrix exponential is formally defined as

$$\exp(R) = I + \sum_{j=1}^{\infty} R^i/i!$$

The computation is nicely solved by the expm package *if* we didn't need derivatives and/or high speed. We want both.

For the package there are three cases:

1. If there is only one departure state, then there is a fast closed form solution, shown below. This case occurs whenever an event time is unique, i.e., no other event times are tied with this one. It also holds, by definition, for competing risk models.

2. If the rate matrix R is upper triangular and the (non-zero) diagonal elements are distinct, there is a fast matrix decomposition algorithm. If the transition matrix is acylic then it can be rearranged to be in upper triangular form. The decomposition also gives a simple expression for the derivative.

3. In the general case we use a Pade-Laplace algorithm: the same found in the matexp package.

For a rate matrix R, R_{jk} is the rate of transition from state j to state k, and is itself an exponential $R_{jk} = \exp(\eta_{jk})$. Thus all non-diagonal values must be $/ge0$. Transitions that do not occur have rate 0. The diagonal element is determined by the constraint that row sums are 0. Let $A = \exp(R)$. Also be aware that $\exp(A)\exp(B) \neq \exp(A + B)$ for the case of matrices.

If there is only one non-zero diagonal element, R_{jj} say, then

$$\begin{aligned}
A_{jj} &= e^{R_{jj}} \\
A_{jk} &= \left(1 - e^{R_{jj}}\right) \frac{R_{jk}}{/} \sum_{l \neq j} R_{jl} \\
A_{kk} &= 1; k \neq j
\end{aligned}$$

and all other elements of A are zero. The derivative of A with respect to η_{jk} will be 0 for all rows except row j.

$$\begin{aligned}
\frac{\partial A_{jj}}{\partial \eta_{jk}} &= \frac{\partial \exp(-\sum_{k!=j} \eta_{jk})}{\partial \eta_{jk}} \\
&= -\eta_{jk} A_{jj} \\
\frac{\partial A_{jk}}{\partial \eta_{jk}} &= eta_{jk} A_{jj} \text{ single event type} \\
\frac{\partial A_{jk}}{\partial \eta_{jm}} &= A_{jj} eta_{jm} \frac{R_{jm}}{\sum_{l \neq j} R_{jl}} + (A_{jj} - 1) \frac{\eta_{jm}(1 - \sum_{l \neq j} R_{jl})}{(\sum_{l \neq j} R_{jl})^2}
\end{aligned}$$

If time is continuous then most events will be at a unique event time, and this fast computation will be the most common case.

If the state space is acylic, the case for many survival problems, then we can reorder the states so that R is upper triangular. In that case, the diagonal elements of R are the eigenvalues. If these are unique (ignoring the zeros), then an algorithm of Kalbfleisch and Lawless gives both A and the derivatives of A in terms of a matrix decomposition. For the remaining cases use the Pade' approximation as found in the matexp package. The overall stategy is the following:

1. Call `survexpmsetup` once, which will decide if the matrix is acyclic, and return a reorder vector if so or a flag if it is not. This determination is based on the possible transitions, e.g., on the transitions matrix from survcheck.
2. Call `survexpm` for each individual transition matrix. In that routine
   - First check for the simple case, otherwise
   - Do not need derivatives: call survexpm
   - Do need derivatives
     - If upper triangular and no tied values, use the deriv routine
     - Otherwise use the Pade routine

⟨*survexpm*⟩=
```
survexpmsetup <- function(rmat) {
    # check the validity of the transition matrix, and determine if it
    #  is acyclic, i.e., can be reordered into an upper triangular matrix.
    if (!is.matrix(rmat) || nrow(rmat) != ncol(rmat) || any(diag(rmat) > 0) ||
        any(rmat[row(rmat) != col(rmat)] < 0))
        stop ("input is not a transition matrix")
    if (!is.logical(all.equal(rowSums(rmat), rep(0, ncol(rmat)))))
        stop("input is not a transition matrix")
    nc <- ncol(rmat)
    lower <- row(rmat) > col(rmat)
    if (all(rmat[lower] ==0))  return(0)  # already in order

    # score each state by (number of states it follows) - (number it precedes)
    temp <- 1*(rmat >0) # 0/1 matrix
    indx <- order(colSums(temp) - rowSums(temp))
    temp <- rmat[indx, indx]  # try that ordering
    if (all(temp[lower]== 0)) indx  # it worked!
    else -1  # there is a loop in the states
}
```

10.1 Decompostion

Based on Kalbfleisch and Lawless, “The analysis of panel data under a Markov assumption” (J Am Stat Assoc, 1985:863-871), the rate matrix R can be written as ADA^{-1} for some matrix A, where D is a diagonal matrix of eigenvalues, provided all of the eigenvalues are distinct. Then $R^k = AD^kA^{-1}$, and using the definition of a matrix exponential we see that $\exp(R) = A\exp(D)A^{-1}$. The exponential of a diagonal matrix is simply a diagonal matrix of the exponentials. The matrix Rt for a scalar t has decomposition $A\exp(Dt)A^{-1}$; a single decompostion suffices for all values of t.

A particular example is

$$R = \begin{pmatrix} r_{11} & r_{12} & r_{13} & 0 & 0 & r_{15} \\ 0 & r_{22} & 0 & r_{24} & 0 & r_{25} \\ 0 & 0 & r_{33} & r_{34} & r_{35} & r_{35} \\ 0 & 0 & 0 & r_{44} & r_{45} & r_{45} \\ 0 & 0 & 0 & 0 & r_{55} & r_{55} \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}. \tag{29}$$

Since this is a transition matrix the diagonal elements are constrained so that row sums are zero: $r_{ii} = -\sum_{j \neq i} r_{ij}$. Since R is an upper triangular matrix it’s eigenvalues lie on the diagonal. If none of the the eigenvalues are repeated, then the Prentice result applies.

The decompostion is quite simple since R is triangular. We want the eigenvectors, i.e. solutions to

$$Rv_i = r_{ii}v_i$$

for $i = 1, \ldots, 6$, where v_i are the colums of V.

It turns out that the set of eigenvectors is also upper triangular; we can solve for them one by one using back substitution. For the first eigenvector we have $v_1 = (1, 0, 0, 0, 0, 0)$. For the second we have the equations

$$
\begin{aligned}
r_{11}x + r_{12}y &= r_{22}x \\
r_{22}y &= r_{22}y
\end{aligned}
$$

which has the solution $(r_{12}/(r_{22} - r_{11}), 1, 0, 0, 0, 0)$, and the process recurs for other rows. Since V is triangular the inverse of V is upper triangular and also easy to compute.

This approach fails if there are tied eigenvalues. Kalbfleice and Lawless comment that this case is rare, but one can then use a decomposition to Jordan canonical form re Cox and Miller, the Theory of Stochastic Processes, 1965. Although this leads to some nice theorems it does not give a simple comutational form, however, and it is easier to fall back on the pade routine. At this time, the pade routine is as fast as the triangluar code, at least for small matrices without deriviatives.

```
⟨survexpm⟩=
 survexpm <- function(rmat, time=1.0, setup, eps=1e-6) {
     # rmat is a transition matrix, so the diagonal elements are 0 or negative
     if (length(rmat)==1) exp(rmat[1]*time)  #failsafe -- should never be called
     else {
         nonzero <- (diag(rmat) != 0)
         if (sum(nonzero ==0)) diag(nrow(rmat))  # expm(0 matrix) = identity
         if (sum(nonzero) ==1) {   # only one state had departures
             j <- which(nonzero)
             emat <- diag(nrow(rmat))
             temp <- exp(rmat[j,j] * time)
             emat[j,j] <- temp
             emat[j, -j] <- (1-temp)* rmat[j, -j]/sum(rmat[j,-j])
             emat
         }
         else if (missing(setup) || setup[1] < 0 ||
                  any(diff(sort(diag(rmat)))< eps)) pade(rmat*time)
         else {
             if (setup[1]==0) .Call(Ccdecomp, rmat, time)$P
             else {
                 temp <- rmat
                 temp[setup, setup] <- .Call(Ccdecomp, rmat[setup, setup], time)
                 temp$P
             }
         }
     }
 }
```

The routine below is modeled after the cholesky routines in the survival library. To help with notation, the return values are labeled as in the Kalbfleisch and Lawless paper, except that their Q = our rmat. Q = A diag(d) Ainv and P= exp(Qt)

```
⟨cdecomp⟩=
 /*
```

```
** Compute the eigenvectors for the upper triangular matrix R
*/
#include <math.h>
#include "R.h"
#include "Rinternals.h"

SEXP cdecomp(SEXP R2, SEXP time2) {
    int i,j,k;
    int nc, ii;

    static const char *outnames[]= {"d", "A", "Ainv",
                                    "P", ""};
    SEXP rval, stemp;
    double *R, *A, *Ainv, *P;
    double *dd, temp, *ediag;
    double time;

    nc = ncols(R2);   /* number of columns */
    R = REAL(R2);
    time = asReal(time2);

    /* Make the output matrices as copies of R, so as to inherit
    **   the dimnames and etc
    */

    PROTECT(rval = mkNamed(VECSXP, outnames));
    stemp=  SET_VECTOR_ELT(rval, 0, allocVector(REALSXP, nc));
    dd = REAL(stemp);
    stemp = SET_VECTOR_ELT(rval, 1, allocMatrix(REALSXP, nc, nc));
    A = REAL(stemp);
    for (i =0; i< nc*nc; i++) A[i] =0;   /* R does not zero memory */
    stemp = SET_VECTOR_ELT(rval, 2, duplicate(stemp));
    Ainv = REAL(stemp);
    stemp = SET_VECTOR_ELT(rval, 3, duplicate(stemp));
    P = REAL(stemp);

    ediag = (double *) R_alloc(nc, sizeof(double));

    /*
    **        Compute the eigenvectors
    **   For each column of R, find x such that Rx = kx
    **   The eigenvalue k is R[i,i], x is a column of A
    **  Remember that R is in column order, so the i,j element is in
    **   location i + j*nc
    */
    ii =0; /* contains i * nc */
```

```
for (i=0; i<nc; i++) { /* computations for column i */
    dd[i] = R[i +ii];    /* the i,i diagonal element = eigenvalue*/
    A[i +ii] = 1.0;
    for (j=(i-1); j >=0; j--) {  /* fill in the rest */
        temp =0;
        for (k=j; k<=i; k++) temp += R[j + k*nc]* A[k +ii];
        A[j +ii] = temp/(dd[i]- R[j + j*nc]);
    }
    ii += nc;
}

/*
** Solve for A-inverse, which is also upper triangular. The diagonal
**  of A and the diagonal of A-inverse are both 1.  At the same time
**  solve for P = A D Ainverse, where D is a diagonal matrix
**  with exp(eigenvalues) on the diagonal.
** P will also be upper triangular, and we can solve for it using
**  nearly the same code as above.  The prior block had RA = x with A the
**  unknown and x successive colums of the identity matrix.
**  We have PA = AD, so x is successively columns of AD.
** Imagine P and A are 4x4 and we are solving for the second row
**  of P.  Remember that P[2,1]= A[2,3] = A[2,4] =0; the equations for
**  this row of P are:
**
**    0*A[1,2] + P[2,2]A[2,2] + P[2,3] 0     + P[2,4] 0     = A[2,2] D[2]
**    0*A[1,3] + P[2,2]A[2,3] + P[2,3]A[3,3] + P[2,4] 0     = A[2,3] D[3]
**    0*A[1,4] + P[2,2]A[2,4] + P[2,3]A[3,4] + P[2,4]A[4,4] = A[2,4] D[4]
**
**  For A-inverse the equations are (use U= A-inverse for a moment)
**    0*A[1,2] + U[2,2]A[2,2] + U[2,3] 0     + U[2,4] 0     = 1
**    0*A[1,3] + U[2,2]A[2,3] + U[2,3]A[3,3] + U[2,4] 0     = 0
**    0*A[1,4] + U[2,2]A[2,4] + U[2,3]A[3,4] + U[2,4]A[4,4] = 0
*/

ii =0; /* contains i * nc */
for (i=0; i<nc; i++) ediag[i] = exp(time* dd[i]);
for (i=0; i<nc; i++) {
    /* computations for column i of A-inverse */
    Ainv[i+ii] = 1.0 ;
    for (j=(i-1); j >=0; j--) {  /* fill in the rest of the column*/
        temp =0;
        for (k=j+1; k<=i; k++) temp += A[j + k*nc]* Ainv[k +ii];
        Ainv[j +ii] = -temp;
    }

    /* column i of P */
```

```
        P[i + ii] = ediag[i];
        for (j=0; j<i; j++) {
            temp =0;
            for (k=j; k<nc; k++) temp += A[j + k*nc] * Ainv[k+ii] * ediag[k];
            P[j+ii] = temp;
        }

        /* alternate computations for row i of P, does not use Ainv*/
        /*P[i +ii] = ediag[i];
          for (j=i+1; j<nc; j++) {
              temp =0;
              for (k=i; k<j; k++) temp += P[i+ k*nc]* A[k + j*nc];
              P[i + j*nc] = (A[i + j*nc]*ediag[j] - temp)/A[j + j*nc];
          }
        */
        ii += nc;
    }
    UNPROTECT(1);
    return(rval);
}
```

10.2 Derivatives

From Kalbfliesch and Lawless, the first derivative of $P = \exp(Rt)$ is

$$
\begin{aligned}
\frac{\partial P}{\partial \theta} &= AVA^{-1} \\
V_{ij} &= \left\{ \begin{array}{ll} G_{ij}(e^{d_i t} - e^{d_j t})/(d_i - d_j) & i \neq j \\ G_{ii} t e^{d_i t} & i = j \end{array} \right. \\
G &= A(\partial R/\partial \theta)A^{-1}
\end{aligned}
$$

The formula for the off diagonal elements collapses to give the formula for the diagonal ones by an application of L'Hospital's rule (for the math geeks).

Each off diagonal element of R is $\exp(X_i\beta) = \exp(\eta_i)$ for a fixed vector X_i — we are computing the derivative at a particular trial value. The first derivative with respect to β_j is then $X_{ij}\exp(\eta_i)$. Since the rows of R sum to a constant then the rows of its derivative must sum to zero; we can fill in the diagonal element after the off diagonal ones are computed. This notation has left something out: there is a separate η vector for each of the non-zero transitions, giving a matrix of derivatives (P is a matrix after all) for each β_j.

This computation is more bookkeeping than the earlier one, but no single portion is particularly intensive computationally when the number of states is modest.

The input will be the X matrix row for the particular subject, the coefficient matrix, the rates matrix, time interval, and the mapping vector from eta to the rates. The last tells us where the zeros are.

⟨*survexpm*⟩=
```
derivative <- function(rmat, time, dR, setup, eps=1e-8) {
    if (missing(setup) || setup[1] <0 || any(diff(sort(diag(rmat)))< eps))
        return (pade(rmat*time, dR*time))

    if (setup==0) dlist <- .Call(Ccdecomp, rmat, time)
    else dlist <- .Call(Ccdecomp, rmat[setup, setup], time)
    ncoef <- dim(dR)[3]
    nstate <- nrow(rmat)

    dmat <- array(0.0, dim=c(nstate, nstate, ncoef))
    vtemp <- outer(dlist$d, dlist$d,
                   function(a, b) {
                       ifelse(abs(a-b)< eps, time* exp(time* (a+b)/2),
                         (exp(a*time) - exp(b*time))/(a-b))})

    # two transitions can share a coef, but only for the same X variable
    for (i in 1:ncoef) {
        G <- dlist$Ainv %*% dR[,,i] %*% dlist$A
        V <- G*vtemp
        dmat[,,i] <- dlist$A %*% V %*% dlist$Ainv
    }
    dlist$dmat <- dmat

    # undo the reordering, if needed
    if (setup[1] >0) {
        indx <- order(setup)
        dlist <- list(P = dlist$P[indx, indx],
                      dmat = apply(dmat,1:2, function(x) x[indx, indx]))
    }

    dlist
}
```

The Pade approximation is found in the file pade.R. There is a good discussion of the problem at www.maths.manchester.ac.uk/ higham/talks/exp09.pdf. The pade function copied code from the matexp package, which in turn is based on Higham 2005. Let B be a matrix and define

$$\begin{aligned} r_m(B) &= p(B)/q(B) \\ p(B) &= \sum_{j=0^m} \frac{((2m-j)!m!}{(2m)!(m-j)!j!} B^j \\ q(B) &= p(-B) \end{aligned}$$

The algorithm for calculating $\exp(A)$ is based on the following table

$\|\|A\|\|_1$	0.15	.25	.95	2.1	3.4
m	3	5	7	9	13

The 1 norm of a matrix is `max(colSums(A))`. If the norm is ≤ 3.4 the $\exp(A) = r_m(A)$ using the table. Otherwise, find s such that $B = A/2^s$ has norm <= 3.4 and use the table method to find $\exp(B)$, then $\exp(A) \approx B^{(}2^s)$, the latter involves repeated squaring of the matrix.

The expm code has a lot of extra steps whose job is to make sure that elements of A are not too disparate in size. Transition matrices are nice and we can skip all of that. This makes the pade function conserably faster than the expm function from the Matrix library. In fact, if there aren't any tied event times, most elements of the rate matrix will be zero, and others are on the order of 1/(number at risk), so that $m = 3$ is the most common outcome.

11 Plotting survival curves

This version of the curves uses the newer form of the survfit object, which fixes an original design decision that I now consider to have been a mistake. That is, an ordinary survival curve did not store the intial (time=0, S=1) point in the survfit object, leaving it up to plotting and/or printing routines to glue it back on. Later additions of delayed starting time and multi-state curves meant that I had to store those values anyway, sticking them into appended objects. The version3 survfit object puts the intial time back where it belongs, and makes this routine easier to write.

The plot, lines, and points routines use several common code blocks in order to maintain consistency.

The xmax argument has been a long term issue. Using xmax on a plot call, we would like that xmax to persist in a subsequent lines.survfit call. But, the problem with this is that lines might not be called after plot.survfit: someone might have other data and then want to add a survfit line to it (rare case I know). If we save the xlimits in some global object, there is no way to erase that object every time a high level call is made.

```
⟨plot.survfit⟩=
 plot.survfit<- function(x, conf.int,  mark.time=FALSE,
                         pch=3,  col=1,lty=1, lwd=1,
                         cex=1, log=FALSE,
                         xscale=1, yscale=1,
                         xlim, ylim, xmax,
                         fun, xlab="", ylab="", xaxs='r',
                         conf.times, conf.cap=.005, conf.offset=.012,
                         conf.type=c('log',  'log-log',  'plain',
                                     'logit', "arcsin"),
                         mark, noplot="(s0)", cumhaz=FALSE,
                         firstx, ymin, ...) {

     dotnames <- names(list(...))
     if (any(dotnames =='type'))
         stop("The graphical argument 'type' is not allowed")
     x <- survfit0(x, x$start.time)   # align data at 0 for plotting

     ⟨plot-log⟩
     ⟨plot-data⟩
     ⟨plot-confint⟩
     ⟨plot-transform⟩
```

```
    ⟨plot-setup-marks⟩
    ⟨plot-makebox⟩
    ⟨plot-functions⟩
    type <- 's'
    ⟨plot-draw⟩
    invisible(lastx)
}

lines.survfit <- function(x, type='s',
                          pch=3, col=1, lty=1, lwd=1,
                          cex=1,
                          mark.time=FALSE, xmax,
                          fun,  conf.int=FALSE,
                          conf.times, conf.cap=.005, conf.offset=.012,
                          conf.type=c('log',  'log-log',  'plain',
                                  'logit', "arcsin"),
                          mark, noplot="(s0)", cumhaz=FALSE, ...) {
    x <- survfit0(x, x$start.time)

    xlog <- par("xlog")
    ⟨plot-data⟩
    ⟨plot-confint⟩
    ⟨plot-transform⟩
    ⟨plot-setup-marks⟩

    # remember a prior xmax
    if (missing(xmax)) xmax <- getOption("plot.survfit")$xmax
    ⟨plot-functions⟩
    ⟨plot-draw⟩
    invisible(lastx)
}

points.survfit <- function(x, fun, censor=FALSE,
                           col=1, pch, noplot="(s0)", cumhaz=FALSE, ...) {

    conf.int <- conf.times <- FALSE  # never draw these with 'points'
    x <- survfit0(x, x$start.time)

    ⟨plot-data⟩
    ⟨plot-transform⟩

    if (ncurve==1 || (length(col)==1 && missing(pch))) {
        if (censor) points(stime, ssurv, ...)
        else points(stime[x$n.event>0], ssurv[x$n.event>0], ...)
    }
    else {
        c2 <- 1  #cycles through the colors and characters
        col <- rep(col, length=ncurve)
        if (!missing(pch)) {
            if (length(pch)==1)
                pch2 <- rep(strsplit(pch, '')[[1]], length=ncurve)
            else pch2 <- rep(pch, length=ncurve)
        }
        for (j in 1:ncol(ssurv)) {
            for (i in unique(stemp)) {
                if (censor) who <- which(stemp==i)
                else who <- which(stemp==i & x$n.event >0)
                if (missing(pch))
                    points(stime[who], ssurv[who,j], col=col[c2], ...)
                else
                    points(stime[who], ssurv[who,j], col=col[c2],
                           pch=pch2[c2], ...)
                c2 <- c2+1
            }
        }
    }
}
```

⟨*plot-log*⟩=

```
# decide on logarithmic axes, yes or no
if (is.logical(log)) {
    ylog <- log
    xlog <- FALSE
    if (ylog) logax <- 'y'
    else      logax <- ""
}
else {
    ylog <- (log=='y' || log=='xy')
    xlog <- (log=='x' || log=='xy')
    logax  <- log
}

if (!missing(fun)) {
    if (is.character(fun)) {
        if (fun=='log'|| fun=='logpct') ylog <- TRUE
        if (fun=='cloglog') {
            xlog <- TRUE
            if (ylog) logax <- 'xy'
            else logax <- 'x'
        }
        if (fun=="cumhaz" && missing(cumhaz)) cumhaz <- TRUE
    }
}
```

⟨*plot-data*⟩=

```
# The default for plot and lines is to add confidence limits
#  if there is only one curve
if (missing(conf.int) && missing(conf.times))
    conf.int <- (!is.null(x$std.err) && prod(dim(x) ==1))

if (missing(conf.times)) conf.times <- NULL
else {
    if (!is.numeric(conf.times)) stop('conf.times must be numeric')
    if (missing(conf.int)) conf.int <- TRUE
}

if (!missing(conf.int)) {
    if (is.numeric(conf.int)) {
        conf.level <- conf.int
        if (conf.level<0 || conf.level > 1)
            stop("invalid value for conf.int")
        if (conf.level ==0) conf.int <- FALSE
        else if (conf.level != x$conf.int) {
            x$upper <- x$lower <- NULL  # force recomputation
        }
        conf.int <- TRUE
    }
    else conf.level = 0.95
}

# Organize data into stime, ssurv, supper, slower
stime <- x$time
std   <- NULL
yzero <- FALSE   # a marker that we have an "ordinary survival curve" with min 0
smat <- function(x) {
    # the rest of the routine is simpler if everything is a matrix
    dd <- dim(x)
    if (is.null(dd)) as.matrix(x)
    else if (length(dd) ==2) x
    else matrix(x, nrow=dd[1])
}

if (cumhaz) {  # plot the cumulative hazard instead
    if (is.null(x$cumhaz))
        stop("survfit object does not contain a cumulative hazard")

    if (is.numeric(cumhaz)) {
        dd <- dim(x$cumhaz)
```

```
            if (is.null(dd)) nhazard <- 1
            else nhazard <- prod(dd[-1])

            if (cumhaz != floor(cumhaz)) stop("cumhaz argument is not integer")
            if (any(cumhaz < 1 | cumhaz > nhazard)) stop("subscript out of range")
            ssurv <- smat(x$cumhaz)[,cumhaz, drop=FALSE]
            if (!is.null(x$std.chaz)) std <- smat(x$std.chaz)[,cumhaz, drop=FALSE]
        }
        else if (is.logical(cumhaz)) {
            ssurv <- smat(x$cumhaz)
            if (!is.null(x$std.chaz)) std <- smat(x$std.chaz)
        }
        else stop("invalid cumhaz argument")
    }
    else if (inherits(x, "survfitms")) {
        i <- !(x$states %in% noplot)
        if (all(i) || !any(i)) {
            # the !any is a failsafe, in case none are kept we ignore noplot
            ssurv <- smat(x$pstate)
            if (!is.null(x$std.err)) std <- smat(x$std.err)
            if (!is.null(x$lower)) {
                slower <- smat(x$lower)
                supper <- smat(x$upper)
            }
        }
        else {
            i <- which(i)  # the states to keep
            # we have to be careful about subscripting
            if (length(dim(x$pstate)) ==3) {
                ssurv <- smat(x$pstate[,,i, drop=FALSE])
                if (!is.null(x$std.err))
                    std <- smat(x$std.err[,,i, drop=FALSE])
                if (!is.null(x$lower)) {
                    slower <- smat(x$lower[,,i, drop=FALSE])
                    supper <- smat(x$upper[,,i, drop=FALSE])
                }
            }
            else {
                ssurv <- x$pstate[,i, drop=FALSE]
                if (!is.null(x$std.err)) std <- x$std.err[,i, drop=FALSE]
                if (!is.null(x$lower)) {
                    slower <- smat(x$lower[,i, drop=FALSE])
                    supper <- smat(x$upper[,i, drop=FALSE])
                }
            }
        }
```

```
}
else {
    yzero <- TRUE
    ssurv <- as.matrix(x$surv)   # x$surv will have one column
    if (!is.null(x$std.err)) std <- as.matrix(x$std.err)
    # The fun argument usually applies to single state survfit objects
    #  First deal with the special case of fun='cumhaz', which is here for
    #  backwards compatability; people should use the cumhaz argument
    if (!missing(fun) && is.character(fun) && fun=="cumhaz") {
        cumhaz <- TRUE
        if (!is.null(x$cumhaz)) {
            ssurv <- as.matrix(x$cumhaz)
            if (!is.null(x$std.chaz)) std <- as.matrix(x$std.chaz)
        } 
        else {
            ssurv <- as.matrix(-log(x$surv))
            if (!is.null(x$std.err)) {
                if (x$logse) std <- as.matrix(x$std.err)
                else std <- as.matrix(x$std.err/x$surv)
            }
         }
    }
}

# set up strata
if (is.null(x$strata)) {
    nstrat <- 1
    stemp <- rep(1, length(x$time)) # same length as stime
}
else {
    nstrat <- length(x$strata)
    stemp <- rep(1:nstrat, x$strata) # same length as stime
}
ncurve <- nstrat * ncol(ssurv)
```

If confidence limits are to be plotted, and they were not part of the data that is passed in, create them. Confidence limits for the cumulative hazard must always be created, and they don't use transforms.

⟨*plot-confint*⟩=
```
conf.type <- match.arg(conf.type)
if (conf.type=="none") conf.int <- FALSE
if (conf.int== "none") conf.int <- FALSE
if (conf.int=="only") {
    plot.surv <- FALSE
    conf.int <- TRUE
    }
```

```
 else plot.surv <- TRUE

 if (conf.int) {
     if (is.null(std)) stop("object does not have standard errors, CI not possible")
     if (cumhaz) {
         if (missing(conf.type)) conf.type="plain"
         temp <- survfit_confint(ssurv, std, logse=FALSE,
                                 conf.type, conf.level, ulimit=FALSE)
         supper <- as.matrix(temp$upper)
         slower <- as.matrix(temp$lower)
     }
     else if (is.null(x$upper)) {
         if (missing(conf.type) && !is.null(x$conf.type))
             conf.type <- x$conf.type
         temp <- survfit_confint(ssurv, std, logse= x$logse,
                                 conf.type, conf.level, ulimit=FALSE)
         supper <- as.matrix(temp$upper)
         slower <- as.matrix(temp$lower)
     }
     else if (!inherits(x, "survfitms")) {
         supper <- as.matrix(x$upper)
         slower <- as.matrix(x$lower)
     }
 } else supper <- slower <- NULL
```

The functional form of the fun argument can be whatever the user wants. For the character form we try to thin out the obvious mistakes. If fun=='cumhaz', the code above has already replaced ssurv with the cumulative hazard, so this part of the code should plug in an identity function.

⟨*plot-transform*⟩=

```
 if (!missing(fun)){
     if (is.character(fun)) {
         if (cumhaz) {
             tfun <- switch(tolower(fun),
                            'log' = function(x) x,
                            'cumhaz'=function(x) x,
                            'identity'= function(x) x,
                            stop("Invalid function argument")
                            )
         } else if (inherits(x, "survfitms")) {
             tfun <-switch(tolower(fun),
                           'log' = function(x) log(x),
                           'event'=function(x) x,
                           'cloglog'=function(x) log(-log(1-x)),
                           'cumhaz' = function(x) x,
                           'pct' = function(x) x*100,
                           'identity'= function(x) x,
                           stop("Invalid function argument")
                           )
         } else {
             yzero <- FALSE
             tfun <- switch(tolower(fun),
                        'log' = function(x) x,
                        'event'=function(x) 1-x,
                        'cumhaz'=function(x) x,
                        'cloglog'=function(x) log(-log(x)),
                        'pct' = function(x) x*100,
                        'logpct'= function(x) 100*x,  #special case further below
                        'identity'= function(x) x,
                        'f' = function(x) 1-x,
                        's' = function(x) x,
                        'surv' = function(x) x,
                        stop("Unrecognized function argument")
                        )
         }
     }
     else if (is.function(fun)) tfun <- fun
     else stop("Invalid 'fun' argument")

     ssurv <- tfun(ssurv )
     if (!is.null(supper)) {
         supper <- tfun(supper)
         slower <- tfun(slower)
     }
 }
```

The mark argument is a holdover from S, when pch could not have numeric values; mark has since disappeared from the manual page for par. We honor it for backwards compatability. To be consistent with matplot and others, we allow pch to be a character string or a vector of characters.

```
⟨plot-setup-marks⟩=
if (missing(mark.time) & !missing(mark)) mark.time <- TRUE
if (missing(pch) && !missing(mark)) pch <- mark
if (length(pch)==1 && is.character(pch)) pch <- strsplit(pch, "")[[1]]

# Marks are not placed on confidence bands
pch  <- rep(pch, length.out=ncurve)
mcol <- rep(col, length.out=ncurve)
if (is.numeric(mark.time)) mark.time <- sort(mark.time)

# The actual number of curves is ncurve*3 if there are confidence bands,
#  unless conf.times has been given.  Colors and line types in the latter
#  match the curves
# If the number of line types is 1 and lty is an integer, then use lty
#    for the curve and lty+1 for the CI
# If the length(lty) <= length(ncurve), use the same color for curve and CI
#   otherwise assume the user knows what they are about and has given a full
#   vector of line types.
# Colors and line widths work like line types, excluding the +1 rule.
if (conf.int & is.null(conf.times)) {
    if (length(lty)==1 && is.numeric(lty))
        lty <- rep(c(lty, lty+1, lty+1), ncurve)
    else if (length(lty) <= ncurve)
        lty <- rep(rep(lty, each=3), length.out=(ncurve*3))
    else lty <- rep(lty, length.out= ncurve*3)

    if (length(col) <= ncurve) col <- rep(rep(col, each=3), length.out=3*ncurve)
    else col <- rep(col, length.out=3*ncurve)

    if (length(lwd) <= ncurve) lwd <- rep(rep(lwd, each=3), length.out=3*ncurve)
    else lwd <- rep(lwd, length.out=3*ncurve)
}
else {
    col  <- rep(col, length.out=ncurve)
    lty  <- rep(lty, length.out=ncurve)
    lwd  <- rep(lwd, length.out=ncurve)
}
```

Create the frame for the plot. We draw an empty figure, letting R figure out the limits.

⟨*plot-makebox*⟩=

```
# check consistency
if (!missing(xlim)) {
    if (!missing(xmax)) warning("cannot have both xlim and xmax arguments, xmax ignored")
    if (!missing(firstx)) stop("cannot have both xlim and firstx arguments")
}
if (!missing(ylim)) {
    if (!missing(ymin)) stop("cannot have both ylim and ymin arguments")
}

# Do axis range computations
if (!missing(xlim) && !is.null(xlim)) {
    tempx <- xlim
    xmax <- xlim[2]
    if (xaxs == 'S') tempx[2] <- tempx[1] + diff(tempx)*1.04
}
else {
    temp <-  stime[is.finite(stime)]
    if (!missing(xmax) && missing(xlim)) temp <- pmin(temp, xmax)
    else xmax <- NULL

    if (xaxs==’S’) {
        rtemp <- range(temp)
        delta <- diff(rtemp)
        #special x- axis style for survival curves
        if (xlog) tempx <- c(min(rtemp[rtemp>0]), min(rtemp)+ delta*1.04)
        else tempx <- c(min(rtemp), min(rtemp)+ delta*1.04)
    }
    else if (xlog) tempx <- range(temp[temp > 0])
    else tempx <- range(temp)
}
if (!missing(xlim) || !missing(xmax))
    options(plot.survfit = list(xmax=tempx[2]))
else options(plot.survfit = NULL)

if (!missing(ylim) && !is.null(ylim)) tempy <- ylim
else {
    skeep <- is.finite(stime) & stime >= tempx[1] & stime <= tempx[2]

    if (ylog) {
        if (!is.null(supper))
            tempy <- range(c(slower[is.finite(slower) & slower>0 & skeep],
                             supper[is.finite(supper) & skeep]))
        else tempy <-  range(ssurv[is.finite(ssurv)& ssurv>0 & skeep])
        if (tempy[2]==1) tempy[2] <- .99   # makes for a prettier axis
        if (any(c(ssurv, slower)[skeep] ==0)) {
            tempy[1] <- tempy[1]*.8
            ssurv[ssurv==0] <- tempy[1]
            if (!is.null(slower))  slower[slower==0] <- tempy[1]
        }
    }
    else {
        if (!is.null(supper))
            tempy <- range(c(supper[skeep], slower[skeep]), finite=TRUE, na.rm=TRUE)
        else tempy <- range(ssurv[skeep], finite=TRUE, na.rm= TRUE)
        if (yzero) tempy <- range(c(0, tempy))
    }
}

if (!missing(ymin)) tempy[1] <- ymin

#
# Draw the basic box
```

```
#
temp <- if (xaxs==’S’) ’i’ else xaxs
plot(range(tempx, finite=TRUE, na.rm=TRUE)/xscale,
     range(tempy, finite=TRUE, na.rm=TRUE)*yscale,
     type=’n’, log=logax, xlab=xlab, ylab=ylab, xaxs=temp,...)
if(yscale != 1) {
    if (ylog) par(usr =par("usr") -c(0, 0, log10(yscale), log10(yscale)))
    else par(usr =par("usr")/c(1, 1, yscale, yscale))
}
if (xscale !=1) {
    if (xlog) par(usr =par("usr") -c(log10(xscale), log10(xscale), 0,0))
    else par(usr =par("usr")*c(xscale, xscale, 1, 1))
}
```

The use of `par(usr)` just above is a bit sneaky. I want the lines and points routines to be able to add to the plot, *without* passing them a global parameter that determines the y-scale or forcing the user to repeat it.

The next functions do the actual drawing.

⟨*plot-functions*⟩=

```
# Create a step function, removing redundancies that sometimes occur in
#  curves with lots of censoring.
dostep <- function(x,y) {
    keep <- is.finite(x) & is.finite(y)
    if (!any(keep)) return()  #all points were infinite or NA
    if (!all(keep)) {
        # these won’t plot anyway, so simplify (CI values are often NA)
        x <- x[keep]
        y <- y[keep]
    }
    n <- length(x)
    if (n==1)       list(x=x, y=y)
    else if (n==2)  list(x=x[c(1,2,2)], y=y[c(1,1,2)])
    else {
        # replace verbose horizonal sequences like
        # (1, .2), (1.4, .2), (1.8, .2), (2.3, .2), (2.9, .2), (3, .1)
        # with (1, .2), (.3, .2),(3, .1).
        #  They are slow, and can smear the looks of the line type.
        temp <- rle(y)$lengths
        drops <- 1 + cumsum(temp[-length(temp)])  # points where the curve drops

        #create a step function
        if (n %in% drops) {  #the last point is a drop
            xrep <- c(x[1], rep(x[drops], each=2))
            yrep <- rep(y[c(1,drops)], c(rep(2, length(drops)), 1))
        }
        else {
            xrep <- c(x[1], rep(x[drops], each=2), x[n])
            yrep <- c(rep(y[c(1,drops)], each=2))
        }
        list(x=xrep, y=yrep)
    }
}

drawmark <- function(x, y, mark.time, censor, cex, ...) {
    if (!is.numeric(mark.time)) {
        xx <- x[censor>0]
        yy <- y[censor>0]
        if (any(censor >1)) {  # tied death and censor, put it on the midpoint
            j <- pmax(1, which(censor>1) -1)
            i <- censor[censor>0]
            yy[i>1] <- (yy[i>1] + y[j])/2
        }
    }
    else { #interpolate
        xx <- mark.time
        yy <- approx(x, y, xx, method="constant", f=0)$y
    }
    points(xx, yy, cex=cex, ...)
}
```

The code to draw the lines and confidence bands.

⟨*plot-draw*⟩=

```
c1 <- 1  # keeps track of the curve number
c2 <- 1  # keeps track of the lty, col, etc
xend <- yend <- double(ncurve)
if (length(conf.offset) ==1)
    temp.offset <- (1:ncurve - (ncurve+1)/2)* conf.offset* diff(par("usr")[1:2])
else temp.offset <- rep(conf.offset, length=ncurve) *  diff(par("usr")[1:2])
temp.cap    <-  conf.cap    * diff(par("usr")[1:2])

for (j in 1:ncol(ssurv)) {
    for (i in unique(stemp)) {  #for each strata
        who <- which(stemp==i)

        # if n.censor is missing, then assume any line that does not have an
        #   event would not be present but for censoring, so there must have
        #   been censoring then
        # otherwise categorize is 0= no censor, 1=censor, 2=censor and death
        if (is.null(x$n.censor)) censor <- ifelse(x$n.event[who]==0, 1, 0)
        else censor <- ifelse(x$n.censor[who]==0, 0, 1 + (x$n.event[who] > 0))
        xx <- stime[who]
        yy <- ssurv[who,j]
```

```
if (conf.int) {
    ylower <- (slower[who,j])
    yupper <- (supper[who,j])
}
if (!is.null(xmax) && max(xx) > xmax) {  # truncate on the right
    xn <- min(which(xx > xmax))
    xx <- xx[1:xn]
    yy <- yy[1:xn]
    xx[xn] <- xmax
    yy[xn] <- yy[xn-1]
    if (conf.int) {
        ylower <- ylower[1:xn]
        yupper <- yupper[1:xn]
        ylower[xn] <- ylower[xn-1]
        yupper[xn] <- yupper[xn-1]
    }
}


if (plot.surv) {
    if (type==’s’)
        lines(dostep(xx, yy), lty=lty[c2], col=col[c2], lwd=lwd[c2])
    else lines(xx, yy, type=type, lty=lty[c2], col=col[c2], lwd=lwd[c2])
    if (is.numeric(mark.time) || mark.time)
        drawmark(xx, yy, mark.time, censor, pch=pch[c1], col=mcol[c1],
                 cex=cex)
}
xend[c1] <- max(xx)
yend[c1] <- yy[length(yy)]

if (conf.int && !is.null(conf.times)) {
    # add vertical bars at the specified times
    x2 <- conf.times + temp.offset[c1]
    templow <- approx(xx, ylower, x2,
                      method=’constant’, f=1)$y
    temphigh<- approx(xx, yupper, x2,
                      method=’constant’, f=1)$y
    segments(x2, templow, x2, temphigh,
              lty=lty[c2], col=col[c2], lwd=lwd[c2])
    if (conf.cap>0) {
        segments(x2-temp.cap, templow, x2+temp.cap, templow,
                 lty=lty[c2], col=col[c2], lwd=lwd[c2] )
        segments(x2-temp.cap, temphigh, x2+temp.cap, temphigh,
                  lty=lty[c2], col=col[c2], lwd=lwd[c2])
    }
```

```
        }
        c1 <- c1 +1
        c2 <- c2 +1

        if (conf.int && is.null(conf.times)) {
            if (type == 's') {
                lines(dostep(xx, ylower), lty=lty[c2],
                      col=col[c2],lwd=lwd[c2])
                c2 <- c2 +1
                lines(dostep(xx, yupper), lty=lty[c2],
                      col=col[c2], lwd= lwd[c2])
                c2 <- c2 + 1
            }
            else {
                lines(xx, ylower, lty=lty[c2],
                      col=col[c2],lwd=lwd[c2], type=type)
                c2 <- c2 +1
                lines(xx, yupper, lty=lty[c2],
                      col=col[c2], lwd= lwd[c2], type= type)
                c2 <- c2 + 1
            }
         }

    }
}
lastx <- list(x=xend, y=yend)
```

12 State space figures

The statefig function was written to do "good enough" state space figures quickly and easily. There are certainly figures it can't draw and many figures that can be drawn better, but it accomplishes its purpose. The key argument `layout`, the first, is a vector of numbers. The value (1,3,4,2) for instance has a single state, then a column with 3 states, then a column with 4, then a column with 2. If `layout` is instead a 1 column matrix then do the same from top down. If it is a 2 column matrix then they provided their own spacing.

⟨*statefig*⟩=
```
statefig <- function(layout, connect, margin=.03, box=TRUE,
                     cex=1, col=1, lwd=1, lty=1, bcol= col,
                     acol=col, alwd = lwd, alty= lty, offset=0) {
    # set up an empty canvas
    frame();  # new environment
    par(usr=c(0,1,0,1))
    if (!is.numeric(layout))
        stop("layout must be a numeric vector or matrix")
```

```
     if (!is.matrix(connect) || nrow(connect) != ncol(connect))
         stop("connect must be a square matrix")
     nstate <- nrow(connect)
     dd <- dimnames(connect)
     if (!is.null(dd[[1]])) statenames <- dd[[1]]
     else if (is.null(dd[[2]]))
         stop("connect must have the state names as dimnames")
     else statenames <- dd[[2]]

     # expand out all of the graphical parameters.  This lets users
     #  use a vector of colors, line types, etc
     narrow <- sum(connect!=0)
     acol <- rep(acol, length=narrow)
     alwd <- rep(alwd, length=narrow)
     alty <- rep(alty, length=narrow)

     bcol <- rep(bcol, length=nstate)
     lty  <- rep(lty, length=nstate)
     lwd  <- rep(lwd, length=nstate)

     col <- rep(col, length=nstate)  # text colors

     ⟨statefig-layout⟩
     ⟨statefig-text⟩
     ⟨statefig-arrows⟩

     dimnames(cbox) <- list(statenames, c("x", "y"))
     invisible(cbox)
 }
 ⟨statefig-fun⟩
```

The drawing region is always (0,1) by (0,1). A user can enter their own matrix of coordinates. Otherwise the free space is divided with one portion on each end and 2 portions between boxes. If there were 3 columns for instance they will have x coordinates of 1/6, 1/6 + 1/3, 1/6 + 2/3. Ditto for dividing up the y coordinate. The primary nuisance is that we want to count down from the top instead of up from the bottom. A 1 by 1 matrix is treated as a column matrix.

```
⟨statefig-layout⟩=
 if (is.matrix(layout) && ncol(layout)==2 && nrow(layout) > 1) {
     # the user provided their own
     if (any(layout <0) || any(layout >1))
         stop("layout coordinates must be between 0 and 1")
     if (nrow(layout) != nstate)
         stop("layout matrix should have one row per state")
     cbox <- layout
 }
```

```
else {
    if (any(layout <=0 | layout != floor(layout)))
        stop("non-integer number of states in layout argument")
    space <- function(n) (1:n -.5)/n   # centers of the boxes
    if (sum(layout) != nstate) stop("number of boxes != number of states")
    cbox <- matrix(0, ncol=2, nrow=nstate)  #coordinates will be here
    n <- length(layout)

    ix <- rep(seq(along=layout), layout)
    if (is.vector(layout) || ncol(layout)> 1) { #left to right
        cbox[,1] <- space(n)[ix]
        for (i in 1:n) cbox[ix==i,2] <- 1 -space(layout[i])
    } else { # top to bottom
        cbox[,2] <- 1- space(n)[ix]
        for (i in 1:n) cbox[ix==i,1] <- space(layout[i])
    }
}
```

Write the text out. Compute the width and height of each box. Then compute the margin. The only tricky thing here is that we want the area around the text to *look* the same left-right and up-down, which depends on the geometry of the plotting region.

⟨*statefig-text*⟩=
```
text(cbox[,1], cbox[,2], statenames, cex=cex, col=col)  # write the labels
textwd <- strwidth(statenames, cex=cex)
textht <- strheight(statenames, cex=cex)
temp <- par("pin")   #plot region in inches
dx <- margin * temp[2]/mean(temp)  # extra to add in the x dimension
dy <- margin * temp[1]/mean(temp)  # extra to add in y

if (box) {
    drawbox <- function(x, y, dx, dy, lwd, lty, col) {
        lines(x+ c(-dx, dx, dx, -dx, -dx),
              y+ c(-dy, -dy, dy, dy, -dy), lwd=lwd, lty=lty, col=col)
    }
    for (i in 1:nstate)
        drawbox(cbox[i,1], cbox[i,2], textwd[i]/2 + dx, textht[i]/2 + dy,
                col=bcol[i], lwd=lwd[i], lty=lty[i])
    dx <- 2*dx; dy <- 2*dy   # move arrows out from the box
    }
```

Now for the hard part, which is drawing the arrows. The entries in the connection matrix are 0= no connection or $1 + d$ for $-1 < d < 1$. The connection is an arc that passes from the center of box 1 to the center of box 2, and through a point that is dz units above the midpoint of the line from box 1 to box 2, where $2z$ is the length of that line. For $d = 1$ we get a half circle to the right (with respect to traversing the line from A to B) and for $d = -1$ we get a half circle to the left. If $d = 0$ it is a straight line.

If A and B are the starting and ending points then AB is the chord of a circle. Draw radii from the center to A, B, and through the midpoint c of AB. This last has length dz above the chord and $r - dz$ below where r is the radius. Then we have

$$\begin{aligned} r^2 &= z^2 + (r - dz)^2 \\ 2rdz &= z^2 + (dz)^2 \\ r &= \left[z(1 + d^2)\right]/2d \end{aligned}$$

Be careful with negative d, which is used to denote left-hand arcs.

The angle θ from A to B is the arctan of $B - A$, and the center of the circle is at $C = (A + B)/2 + (r - dz)(\sin\theta, -\cos\theta)$. We then need to draw the arc $C + r(\cos\phi, \sin\phi)$ for some range of angles ϕ. The angles to the centers of the boxes are $\arctan(A - C)$ and $\arctan(B - C)$, but we want to start and end outside the box. It turned out that this is more subtle than I thought. The solution below uses two helper functions `statefigx` and `statefigy`. The first accepts C, r, the range of ϕ values, and a target y value. It returns the angles, within the range, such that the endpoint of the arc has horizontal coordinate x, or an empty vector if none such exists. For an arc there are sometimes two solutions. First calculate the angles for which the arc will strike the horizontal line. If the arc is too short to reach the line then there is no intersection. The return legal angles.

⟨*statefig-fun*⟩=

```
statefigx <- function(x, C, r, a1, a2) {
    temp <-(x - C[1])/r
    if (abs(temp) >1) return(NULL)  # no intersection of the arc and x
    phi <- acos(temp)  # this will be from 0 to pi
    pi <- 3.1415926545898   # in case someone has a variable "pi"
    if (x > C[1]) phi <-  c(phi, pi - phi)
    else          phi <- -c(phi, pi - phi)
    # Add reflection about the X axis, in both forms
    phi <- c(phi, -phi, 2*pi - phi)
    amax <- max(a1, a2)
    amin <- min(a1, a2)
    phi[phi<amax & phi > amin]
}
statefigy <-  function(y, C, r, a1, a2) {
    pi <- 3.1415926545898   # in case someone has a variable named "pi"
    amax <- max(a1, a2)
    amin <- min(a1, a2)
    temp <-(y - C[2])/r
    if (abs(temp) >1) return(NULL)  # no intersection of the arc and y
    phi <- asin(temp)  # will be from -pi/2 to pi/2
    phi <- c(phi, sign(phi)*pi -phi)  # reflect about the vertical
    phi <- c(phi, phi + 2*pi)
    phi[phi<amax & phi > amin]
}
```

⟨*statefig-fun*⟩=

```
 phi <- function(x1, y1, x2, y2, d, delta1, delta2) {
     # d = height above the line
     theta <- atan2(y2-y1, x2-x1)    # angle from center to center
     if (abs(d) < .001) d=.001       # a really small arc looks like a line

     z <- sqrt((x2-x1)^2 + (y2 - y1)^2) /2 # half length of chord
     ab <- c((x1 + x2)/2, (y1 + y2)/2)      # center of chord
     r  <- abs(z*(1 + d^2)/ (2*d))
     if (d >0) C  <- ab + (r - d*z)* c(-sin(theta), cos(theta)) # center of arc
     else      C  <- ab + (r + d*z)* c( sin(theta), -cos(theta))

     a1 <- atan2(y1-C[2], x1-C[1])   # starting angle
     a2 <- atan2(y2-C[2], x2-C[1])   # ending angle
     if (abs(a2-a1) > pi) {
         # a1= 3 and a2=-3, we don't want to include 0
         # nor for a1=-3 and a2=3
         if (a1>0) a2 <- a2 + 2 *pi
         else a1 <- a1 + 2*pi
     }
     if (d > 0) { #counterclockwise
         phi1 <- min(statefigx(x1 + delta1[1], C, r, a1, a2),
                     statefigx(x1 - delta1[1], C, r, a1, a2),
                     statefigy(y1 + delta1[2], C, r, a1, a2),
                     statefigy(y1 - delta1[2], C, r, a1, a2), na.rm=TRUE)
         phi2 <- max(statefigx(x2 + delta2[1], C, r, a1, a2),
                     statefigx(x2 - delta2[1], C, r, a1, a2),
                     statefigy(y2 + delta2[2], C, r, a1, a2),
                     statefigy(y2 - delta2[2], C, r, a1, a2), na.rm=TRUE)
     }
     else { # clockwise
         phi1 <- max(statefigx(x1 + delta1[1], C, r, a1, a2),
                     statefigx(x1 - delta1[1], C, r, a1, a2),
                     statefigy(y1 + delta1[2], C, r, a1, a2),
                     statefigy(y1 - delta1[2], C, r, a1, a2), na.rm=TRUE)
         phi2 <- min(statefigx(x2 + delta2[1], C, r, a1, a2),
                     statefigx(x2 - delta2[1], C, r, a1, a2),
                     statefigy(y2 + delta2[2], C, r, a1, a2),
                     statefigy(y2 - delta2[2], C, r, a1, a2), na.rm=TRUE)
     }

     list(center=C, angle=c(phi1, phi2), r=r)
 }
```

Now draw the arrows, one at a time. I arbitrarily declare that 20 segments is enough for a smooth curve.

⟨*statefig-arrows*⟩=

```
arrow2 <- function(...) arrows(..., angle=20, length=.1)
doline <- function(x1, x2, d, delta1, delta2, lwd, lty, col) {
    if (d==0 && x1[1] ==x2[1]) { # vertical line
        if (x1[2] > x2[2]) # downhill
            arrow2(x1[1], x1[2]- delta1[2], x2[1], x2[2] + delta2[2],
                   lwd=lwd, lty=lty, col=col)
        else arrow2(x1[1], x1[2]+ delta1[2], x2[1], x2[2] - delta2[2],
                    lwd=lwd, lty=lty, col=col)
    }
    else if (d==0 && x1[2] == x2[2]) {  # horizontal line
        if (x1[1] > x2[1])  # right to left
            arrow2(x1[1]-delta1[1], x1[2], x2[1] + delta2[1], x2[2],
                   lwd=lwd, lty=lty, col=col)
        else arrow2(x1[1]+delta1[1], x1[2], x2[1] - delta2[1], x2[2],
                    lwd=lwd, lty=lty, col=col)
    }
    else {
        temp <- phi(x1[1], x1[2], x2[1], x2[2], d, delta1, delta2)
        if (d==0) {
            arrow2(temp$center[1] + temp$r*cos(temp$angle[1]),
                   temp$center[2] + temp$r*sin(temp$angle[1]),
                   temp$center[1] + temp$r*cos(temp$angle[2]),
                   temp$center[2] + temp$r*sin(temp$angle[2]),
                   lwd=lwd, lty=lty, col=col)
        }
        else {
            # approx the curve with 21 segments
            #  arrowhead on the last one
            phi <- seq(temp$angle[1], temp$angle[2], length=21)
            lines(temp$center[1] + temp$r*cos(phi),
                  temp$center[2] + temp$r*sin(phi), lwd=lwd, lty=lty, col=col)
            arrow2(temp$center[1] + temp$r*cos(phi[20]),
                   temp$center[2] + temp$r*sin(phi[20]),
                   temp$center[1] + temp$r*cos(phi[21]),
                   temp$center[2] + temp$r*sin(phi[21]),
                   lwd=lwd, lty=lty, col=col)
        }
    }
}
```

The last arrow bit is the offset. If offset $\neq 0$ and there is a bidirectional arrow between two boxes, and the arc for both of them is identical, then move each arrow just a bit, orthagonal to a segment connecting the middle of the two boxes. If the line goes from (x1, y1) to (x2, y2), then the normal to the line at (x1, x2) is (y2-y1, x1-x2), normalized to length 1. The -1 below (`-offset`) makes the shift obey a left-hand rule: looking down a line segement towards the arrow head, we shift to the left. This makes two horizontal arrows stack in the normal typographical order for chemical reactions, the right facing one above the left facing. A user can use a negative value for offset to reverse this if they wish.

```
⟨statefig-arrows⟩=
 k <- 1
 for (j in 1:nstate) {
     for (i in 1:nstate) {
         if (i != j && connect[i,j] !=0) {
             if (connect[i,j] == 2-connect[j,i] && offset!=0) {
                 #add an offset
                 toff <- c(cbox[j,2] - cbox[i,2], cbox[i,1] - cbox[j,1])
                 toff <- -offset *toff/sqrt(sum(toff^2))
                 doline(cbox[i,]+toff, cbox[j,]+toff, connect[i,j]-1,
                        delta1 = c(textwd[i]/2 + dx, textht[i]/2 + dy),
                        delta2 = c(textwd[j]/2 + dx, textht[j]/2 + dy),
                        lty=alty[k], lwd=alwd[k], col=acol[k])
                 }
             else doline(cbox[i,], cbox[j,], connect[i,j]-1,
                         delta1 = c(textwd[i]/2 + dx, textht[i]/2 + dy),
                         delta2 = c(textwd[j]/2 + dx, textht[j]/2 + dy),
                         lty=alty[k], lwd=alwd[k], col=acol[k])
             k <- k +1
         }
     }
 }
```

13 tmerge

The tmerge function was designed around a set of specific problems. The idea is to build up a time dependent data set one endpoint at at time. The primary arguments are

- data1: the base data set that will be added onto
- data2: the source for new information
- id: the subject identifier in the new data
- . . . : additional arguments that add variables to the data set
- tstart, tstop: used to set the time range for each subject
- options

The created data set has three new variables (at least), which are `id`, `tstart` and `tstop`.

The key part of the call are the “. . . ” arguments which each can be one of four types: tdc() and cumtdc() add a time dependent variable, event() and cumevent() add a new endpoint. In the survival routines time intervals are open on the left and closed on the right, i.e., (tstart, tstop]. Time dependent covariates apply from the start of an interval and events occur at the end of an interval. If a data set already had intervals of (0,10] and (10, 14] a new time dependent covariate or event at time 8 would lead to three intervals of (0,8], (8,10], and (10,14]; the new time-dependent covariate value would be added to the second interval, a new event would be added to the first one.

A typical call would be

```
⟨dummy⟩=
  newdata <- tmerge(newdata, old, id=clinic, diabetes=tdc(diab.time))
```

which would add a new time dependent covariate diabetes to the data set.

```
⟨tmerge⟩=
 tmerge <- function(data1, data2, id, ..., tstart, tstop, options) {
     Call <- match.call()
     # The function wants to recognize special keywords in the
     #  arguments, so define a set of functions which will be used to
     #  mark objects
     new <- new.env(parent=parent.frame())
     assign("tdc", function(time, value=NULL, init=NULL) {
         x <- list(time=time, value=value, default= init);
         class(x) <- "tdc"; x},
            envir=new)
     assign("cumtdc", function(time, value=NULL, init=NULL) {
         x <- list(time=time, value=value, default= init);
         class(x) <-"cumtdc"; x},
            envir=new)
     assign("event", function(time, value=NULL, censor=NULL) {
         x <- list(time=time, value=value, censor=censor);
         class(x) <-"event"; x},
            envir=new)
     assign("cumevent", function(time, value=NULL, censor=NULL) {
         x <- list(time=time, value=value, censor=censor);
         class(x) <-"cumevent"; x},
            envir=new)

     if (missing(data1) || missing(data2) || missing(id))
         stop("the data1, data2, and id arguments are required")
     if (!inherits(data1, "data.frame")) stop("data1 must be a data frame")
     ⟨tmerge-setup⟩
     ⟨tmerge-addvar⟩
     ⟨tmerge-finish⟩
 }
 ⟨tmerge-print⟩
```

The program can't use formulas because the ... arguments need to be named. This results in a bit of evaluation magic to correctly assess arguments. The routine below could have been set out as a separate top-level routine, the argument is where we want to document it: within the tmerge page or on a separate one. I decided on the former.

⟨*tmerge-setup*⟩=

```
tmerge.control <- function(idname="id", tstartname="tstart", tstopname="tstop",
                           delay =0, na.rm=TRUE, tdcstart=NA_real_, ...) {
    extras <- list(...)
    if (length(extras) > 0)
        stop("unrecognized option(s):", paste(names(extras), collapse=', '))
    if (length(idname) != 1 || make.names(idname) != idname)
        stop("idname option must be a valid variable name")
    if (!is.null(tstartname) &&
        (length(tstartname) !=1 || make.names(tstartname) != tstartname))
        stop("tstart option must be NULL or a valid variable name")
    if (length(tstopname) != 1 || make.names(tstopname) != tstopname)
        stop("tstop option must be a valid variable name")
    if (length(delay) !=1 || !is.numeric(delay) || delay < 0)
        stop("delay option must be a number >= 0")
    if (length(na.rm) !=1 || ! is.logical(na.rm))
        stop("na.rm option must be TRUE or FALSE")
    if (length(tdcstart) !=1) stop("tdcstart must be a single value")
    list(idname=idname, tstartname=tstartname, tstopname=tstopname,
         delay=delay, na.rm=na.rm, tdcstart=tdcstart)
}

if (!inherits(data1, "tmerge") && !is.null(attr(data1, "tname"))) {
    # old style object that someone saved!
    tm.retain <- list(tname = attr(data1, "tname"),
                      tevent= list(name=attr(data1, "tevent"),
                                   censor= attr(data1, "tcensor")),
                      tdcvar = attr(data1, "tdcvar"),
                      n = nrow(data1))
    attr(data1, "tname") <- attr(data1, "tevent") <- NULL
    attr(data1, "tcensor") <- attr(data1, "tdcvar") <- NULL
    attr(data1, "tm.retain") <- tm.retain
    class(data1) <- c("tmerge", class(data1))
}

if (inherits(data1, "tmerge")) {
    tm.retain <- attr(data1, "tm.retain")
    firstcall <- FALSE
    # check out whether the object looks legit:
    #  has someone tinkered with it?  This won't catch everything
    tname <- tm.retain$tname
    tevent <- tm.retain$tevent
    tdcvar <- tm.retain$tdcvar
    if (nrow(data1) != tm.retain$n)
        stop("tmerge object has been modified, size")
    if (any(is.null(match(unlist(tname), names(data1)))) ||
        any(is.null(match(tm.retain$tcdname, names(data1)))) ||
        any(is.null(match(tevent$name, names(data1)))))
        stop("tmerge object has been modified, missing variables")
    for (i in seq(along=tevent$name)) {
        ename <- tevent$name[i]
        if (is.numeric(data1[[ename]])) {
            if (!is.numeric(tevent$censor[[i]]))
                stop("event variable ", ename,
                     " no longer matches it's original class")
        }
        else if (is.character(data1[[ename]])) {
            if (!is.character(tevent$censor[[i]]))
                stop("event variable ", ename,
                     " no longer matches it's original class")
        }
        else if (is.logical(data1[[ename]])) {
            if (!is.logical(tevent$censor[[i]]))
                stop("event variable ", ename,
                     " no longer matches it's original class")
        }
        else if (is.factor(data1[[ename]])) {
            if (levels(data1[[ename]])[1] != tevent$censor[[i]])
                stop("event variable ", ename,
                     " has a new first level")
        }
        else stop("event variable ", ename, " is of an invalid class")
    }
} else {
    firstcall <- TRUE
    tname <- tevent <- tdcvar <- NULL
    if (is.name(Call[["id"]])) {
        idx <- as.character(Call[["id"]])
        if (missing(options)) options <-list(idname= idx)
        else if (is.null(options$idname)) options$idname <- idx
    }
}

if (!missing(options)) {
    if (!is.list(options)) stop("options must be a list")
    if (!is.null(tname)) {
        # If an option name matches one already in tname, don't confuse
        #  the tmerge.control routine with duplicate arguments
        temp <- match(names(options), names(tname), nomatch=0)
        topt <- do.call(tmerge.control, c(options, tname[temp==0]))
        if (any(temp >0)) {
            # A variable name is changing midstream, update the
            # variable names in data1
            varname <- tname[c("idname", "tstartname", "tstopname")]
            temp2 <- match(varname, names(data1))
            names(data1)[temp2] <- varname
        }
    }
    else topt <- do.call(tmerge.control, options)
}
else if (length(tname)) topt <- do.call(tmerge.control, tname)
else  topt <- tmerge.control()

# id, tstart, tstop are found in data2
if (missing(id)) stop("the id argument is required")
if (missing(data1) || missing(data2))
    stop("two data sets are required")
id <- eval(Call[["id"]], data2, enclos=emptyenv()) #don't find it elsewhere
if (is.null(id)) stop("id variable not found in data2")
if (any(is.na(id))) stop("id variable cannot have missing values")

if (firstcall) {
    if (!missing(tstop)) {
         tstop <-  eval(Call[["tstop"]],  data2)
         if (length(tstop) != length(id))
             stop("tstop and id must be the same length")
         # The neardate routine will check for legal tstop data type
      }
    if (!missing(tstart)) {
        tstart <- eval(Call[["tstart"]], data2)
        if (length(tstart)==1) tstart <- rep(tstart, length(id))
        if  (length(tstart) != length(id))
            stop("tstart and id must be the same length")
        if (any(tstart >= tstop))
            stop("tstart must be < tstop")
         }
}
else {
    if (!missing(tstart) || !missing(tstop))
        stop("tstart and tstop arguments only apply to the first call")
}
```

Get the ... arguments. They are evaluated in a special frame, set up earlier, so that the definitions of the functions tdc, cumtdc, event, and cumevent are local to tmerge. Check that they are all legal: each argument is named, and is one of the four allowed types.

⟨*tmerge-setup*⟩=
```
# grab the... arguments
notdot <- c("data1", "data2", "id", "tstart", "tstop", "options")
```

```
dotarg <- Call[is.na(match(names(Call), notdot))]
dotarg[[1]] <- as.name("list")  # The as-yet dotarg arguments
if (missing(data2)) args <- eval(dotarg, envir=new)
else  args <- eval(dotarg, data2, enclos=new)

argclass <- sapply(args, function(x) (class(x))[1])
argname <- names(args)
if (any(argname== "")) stop("all additional argments must have a name")

check <- match(argclass, c("tdc", "cumtdc", "event", "cumevent"))
if (any(is.na(check)))
    stop(paste("argument(s)", argname[is.na(check)],
               "not a recognized type"))
```

The tcount matrix keeps track of what we have done, and is added to the final object at the end. This is useful to the user for debugging what may have gone right or wrong in their usage.

⟨*tmerge-setup*⟩=
```
# The tcount matrix is useful for debugging
tcount <- matrix(0L, length(argname), 9)
dimnames(tcount) <- list(argname, c("early","late", "gap", "within",
                                    "boundary", "leading", "trailing",
                                    "tied", "missid"))
tcens <- tevent$censor
tevent <- tevent$name
if (is.null(tcens)) tcens <- vector('list', 0)
```

The very first call to the routine is special, since this is when the range of legal times is set. We also apply an initial sort to the data if necessary so that times are in order. There are 2 cases:

1. Adding a time range: tstop comes from data2, optional tstart, and the id can be simply matched, by which we mean no duplicates in data1.
2. The more common case: there is no tstop, one observation per subject, and the first optional argument is an event or cumevent. We then use its time as the range.

One thing we could add, but didn't, was to warn if any of the three new variables will stomp on ones already in data1.

Note that in case 2 we cannot wait for the later code to deal with duplicate id/time pairs, since that later code requires a valid starting point. That code will work out which of a duplicate should be retained, however.

⟨*tmerge-setup*⟩=
```
newdata <- data1 #make a copy
if (firstcall) {
    # We don't look for topt$id.  What if the user had id=clinic, but their
    #  starting data set also had a variable named "id".  We want clinic for
    #  this first call.
    idname <- Call[["id"]]
    if (!is.name(idname))
        stop("on the first call 'id' must be a single variable name")

    # The line below finds tstop and tstart variables in data1
    indx <- match(c(topt$idname, topt$tstartname, topt$tstopname), names(data1),
                  nomatch=0)
    if (any(indx[1:2]>0) && FALSE) {  # warning currently turned off. Be chatty?
        overwrite <- c(topt$tstartname, topt$tstopname)[indx[2:3]]
        warning("overwriting data1 variables", paste(overwrite, collapse=' '))
        }

    temp <- as.character(idname)
    if (!is.na(match(temp, names(data1)))) {
            data1[[topt$idname]] <- data1[[temp]]
            baseid <- data1[[temp]]
            }
    else stop("id variable not found in data1")

    if (any(duplicated(baseid)))
        stop("for the first call (that establishes the time range) data1 must have no duplicate id

    if (missing(tstop)) {
        if (length(argclass)==0 || argclass[1] != "event")
            stop("neither a tstop argument nor an initial event argument was found")
        # this is case 2 -- the first time value for each obs sets the range
        last <- !duplicated(id)
        indx2 <- match(unique(id[last]), baseid)
        if (any(is.na(indx2)))
            stop("setting the range, and data2 has id values not in data1")
        if (any(is.na(match(baseid, id))))
            stop("setting the range, and data1 has id values not in data2")
        newdata <- data1[indx2,]
        tstop <- (args[[1]]$time)[last]
    }
    else {
        if (length(baseid)== length(id) && all(baseid == id)) newdata <- data1
        else {  # Note: 'id' is the idlist for data 2
            indx2 <- match(id, baseid)
            if (any(is.na(indx2)))
                stop("setting the range, and data2 has id values not in data1")
            if (any(is.na(match(baseid, id))))
                stop("setting the range, and data1 has id values not in data2")
            newdata <- data1[indx2,]
        }
```

```
     }

     if (any(is.na(tstop)))
         stop("missing time value, when that variable defines the span")
     if (missing(tstart)) {
         indx <- which(tstop <=0)
         if (length(indx) >0) stop("found an ending time of ", tstop[indx[1]],
                                   ", the default starting time of 0 is invalid")
         tstart <- rep(0, length(tstop))
     }
     if (any(tstart >= tstop))
         stop("tstart must be < tstop")
     newdata[[topt$tstartname]] <- tstart
     newdata[[topt$tstopname]] <- tstop
     n <- nrow(newdata)
     if (any(duplicated(id))) {
         # sort by time within id
         indx1 <- match(id, unique(id))
         newdata <- newdata[order(indx1, tstop),]
      }
     temp <- newdata[[topt$idname]]
     if (any(tstart >= tstop)) stop("tstart must be < tstop")
     if (any(newdata$tstop[-n] > newdata$tstart[-1] &
             temp[-n] == temp[-1]))
         stop("first call has created overlapping or duplicated time intervals")
     idmiss <- 0  # the tcount table should have a zero
 }
 else { #not a first call
     idmatch <- match(id, data1[[topt$idname]], nomatch=0)
     if (any(idmatch==0)) idmiss <- sum(idmatch==0)
     else idmiss <- 0
 }
```

Now for the real work. For each additional argument we first match the id/time pairs of the new data to the current data set, and categorize each into a type. If the time value in data2 is NA, then that addition is skipped. Ditto if the value is NA and options narm=TRUE. This is a convenience for the user, who will often be merging in a variable like "day of first diabetes diagnosis" which is missing for those who never had that outcome occur.

⟨*tmerge-addvar*⟩=

```
 saveid <- id
 for (ii in seq(along.with=args)) {
     argi <- args[[ii]]
     baseid <- newdata[[topt$idname]]
     dstart <- newdata[[topt$tstartname]]
     dstop  <- newdata[[topt$tstopname]]
     argcen <- argi$censor
```

```
# if an event time is missing then skip that obs.  Also toss obs that
#  whose id does not match anyone in data1
etime <- argi$time
if (idmiss ==0) keep <- rep(TRUE, length(etime))
else keep <- (idmatch > 0)
if (length(etime) != length(saveid))
    stop("argument ", argname[ii], " is not the same length as id")
if (!is.null(argi$value)) {
   if (length(argi$value) != length(saveid))
        stop("argument ", argname[ii], " is not the same length as id")
    if (topt$na.rm) keep <- keep & !(is.na(etime) | is.na(argi$value))
    else keep <- keep & !is.na(etime)
    if (!all(keep)) {
        etime <- etime[keep]
        argi$value <- argi$value[keep]
        }
    }
else {
  keep <- keep & !is.na(etime)
  etime <- etime[keep]
  }
id <- saveid[keep]

# Later steps become easier if we sort the new data by id and time
#  The match() is critical when baseid is not in sorted order.  The
#  etime part of the sort will change from one ii value to the next.
indx <- order(match(id, baseid), etime)
id <- id[indx]
etime <- etime[indx]
if (!is.null(argi$value))
    yinc <- argi$value[indx]
else yinc <- NULL

# indx1 points to the closest start time in the baseline data (data1)
#  that is <= etime.  indx2 to the closest end time that is >=etime.
# If etime falls into a (tstart, tstop) interval, indx1 and indx2
#   will match
# If the "delay" argument is set and this event is of type tdc, then
#   move any etime that is after the entry time for a subject.
if (topt$delay >0 && argclass[ii] %in% c("tdc", "cumtdc")) {
    mintime <- tapply(dstart, baseid, min)
    index <- match(id, names(mintime))
    etime <- ifelse(etime <= mintime[index], etime, etime+ topt$delay)
}
```

```
    indx1 <- neardate(id, baseid, etime, dstart, best="prior")
    indx2 <- neardate(id, baseid, etime, dstop, best="after")

    # The event times fall into one of 5 categories
    #   1. Before the first interval
    #   2. After the last interval
    #   3. Outside any interval but with time span, i.e, it falls into
    #       a gap in follow-up
    #   4. Strictly inside an interval (does't touch either end)
    #   5. Inside an interval, but touching.
    itype <- ifelse(is.na(indx1), 1,
                    ifelse(is.na(indx2), 2,
                           ifelse(indx2 > indx1, 3,
                                  ifelse(etime== dstart[indx1] |
                                         etime== dstop[indx2], 5, 4))))

    # Subdivide the events that touch on a boundary
    #  1: intervals of (a,b] (b,d], new count at b  "tied edge"
    #  2: intervals of (a,b] (c,d] with c>b, new count at c, "front edge"
    #  3: intervals of (a,b] (c,d] with c>b, new count at b, "back edge"
    #
    subtype <- ifelse(itype!=5, 0,
                      ifelse(indx1 == indx2+1, 1,
                             ifelse(etime==dstart[indx1], 2, 3)))
    tcount[ii,1:7] <- table(factor(itype+subtype, levels=c(1:4, 6:8)))

    # count ties.  id and etime are not necessarily sorted
    tcount[ii,8] <- sum(tapply(etime, id, function(x) sum(duplicated(x))))
    tcount[ii,9] <- idmiss
    ⟨tmerge-addin2⟩
}
```

A tdc or cumtdc operator defines a new time-dependent variable which applies to all future times. Say that we had the following scenario for one subject

current		addition	
tstart	tstop	time	x
2	5	1	20.2
6	7	7	11
7	15	8	17.3
15	30		

The resulting data set will have intervals of (2,5), (6,7), (7,8) and (8,15) with covariate values of 20.2, 20.2, 11, and 17.3. Only a covariate change that occurs within an interval causes a new data row. Covariate changes that happen after the last interval are ignored, i.e. at change at time ≥ 30 in the above example.

If instead this had been events at times 1, 7, and 8, the first event would be ignored since it happens outside of any interval, so would an event at exactly time 2. The event at time 7 would be recorded in the (6,7) interval and the one at time 8 in the (7,8) interval: events happen at the ends of intervals. In both cases new rows are only generated for new time values that fall strictly within one of the old intervals.

When a subject has two increments on the same day the later one wins. This is correct behavior for cumtdc, a bit odd for cumevent, and the user's problem for tdc and event. We report back the number of ties so that the user can deal with it.

Where are we now with the variables?

itype	class	indx1	indx2
1	before	NA	next interval
2	after	prior interval	NA
3	in a gap	prior interval	next interval
4	within interval	containing interval	containing interval
5-1	on a join	next interval	prior interval
5-2	front edge	containing	containing
5-3	back edge	containing	containing

If there are any itype 4, start by expanding the data set to add new cut points, which will turn all the 4's into 5-1 types. When expanding, all the event type variables turn into "censor" at the newly added times and other variables stay the same. A subject could have more than one new cutpoint added within an interval so we have to count each. In newdata all the rows for a given subject are contiguous and in time order, though the data set may not be in subject order.

```
⟨tmerge-addin2⟩=
 indx4 <- which(itype==4)
 n4 <- length(indx4)
 if (n4 > 0) {
     # we need to eliminate duplicate times within the same id, but
     #  do so without changing the class of etime: it might
     #  be a Date, an integer, a double, ...
     # Using unique on a data.frame does the trick
     icount <- data.frame(irow= indx1[indx4], etime=etime[indx4])
     icount <- unique(icount)
     # the icount data frame will be sorted by second column within first
     #  so rle is faster than table
     n.add <- rle(icount$irow)$length # number of rows to add for each id

     # expand the data
     irep <- rep.int(1L, nrow(newdata))
     erow <- unique(indx1[indx4])   # which rows in newdata to be expanded
     irep[erow] <- 1+ n.add # number of rows in new data
     jrep <- rep(1:nrow(newdata), irep)  #stutter the duplicated rows
     newdata <- newdata[jrep,]  #expand it out
     dstart <- dstart[jrep]
     dstop <-  dstop[jrep]

     #fix up times
```

```
    nfix <- length(erow)
    temp <- vector("list", nfix)
    iend <- (cumsum(irep))[irep >1]  #end row of each duplication set
    for (j in 1:nfix) temp[[j]] <-  -(seq(n.add[j] -1, 0)) + iend[j]
    newrows <- unlist(temp)

    # this should not be necessary
    #    if (inherits(dstart, "Date"))
    #        icount$etime <- as.Date(icount$etime, origin= "1970-01-01")
    dstart[newrows] <- dstop[newrows-1] <- icount$etime
    newdata[[topt$tstartname]] <- dstart
    newdata[[topt$tstopname]]  <- dstop
    for (ename in tevent) newdata[newrows-1, ename] <- tcens[[ename]]

    # refresh indices
    baseid <- newdata[[topt$idname]]
    indx1 <- neardate(id, baseid, etime, dstart, best="prior")
    indx2 <- neardate(id, baseid, etime, dstop, best="after")
    subtype[itype==4] <- 1  #all the "insides" are now on a tied edge
    itype[itype==4]   <- 5
}
```

Now we can add the new variable. The most common is a tdc, so start with it. The C routine returns a set of indices: 0,1,1,2,3,0,4,... would mean that row 1 of the new data happens before the tdc variable, 2 and 3 take values from the first element of yinc, etc. By returning an index, the yinc variable can be of any data type. Using is.na() on the left side below causes the *right* kind of NA to be inserted (this trick was stolen from the merge routine).

If this is a first call, don't allow the new variable to overwrite a variable already existing in the data set, we found it leads to problems. (Usually it is a user mistake.) However, tdc calls themselves can stack.

⟨*tmerge-addin2*⟩=
```
# add a tdc variable
newvar <- newdata[[argname[ii]]]  # prior value (for sequential tmerge calls)
if (argclass[ii] %in% c("tdc", "cumtdc")){
    if (argname[[ii]] %in% tevent)
        stop("attempt to turn event variable", argname[[ii]], "into a tdc")
    if (!(argname[[ii]] %in% tdcvar)){
        tdcvar <- c(tdcvar, argname[[ii]])
        if (!is.null(newvar) && argclass[ii] == "tdc") {
            warning(paste0("replacement of variable ’", argname[ii], "’"))
            newvar <- NULL
        }
    }
}
if (argclass[ii] == "tdc") {
    default <- argi$default   # default value
```

```
if (is.null(default)) default <- topt$tdcstart
else if (length(default) !=1)
    stop("default tdc value must be of length 1")

# id can be any data type; feed integers to the C routine
storage.mode(dstart) <- storage.mode(etime) <- "double"  #if time is integer
uid <- unique(baseid)
index <- .Call(Ctmerge2, match(baseid, uid), dstart,
                           match(id, uid),  etime)

if (is.null(newvar)) {  # create new variable
    if (is.null(yinc)) newvar <- ifelse(index==0, 0L, 1L) #add a 0/1 variable
    else {
        newvar <- yinc[pmax(1L, index)]
        if (any(index==0)) {
            if (is.na(default)) is.na(newvar) <- (index==0L)
            else {
                if (is.numeric(newvar)) newvar[index==0L] <- as.numeric(default)
                else {
                    if (is.factor(newvar)) {
                        # special case: if default isn't in the set of levels,
                        #   add it to the levels
                        if (is.na(match(default, levels(newvar))))
                            levels(newvar) <- c(levels(newvar), default)
                    }
                    newvar[index== 0L] <- default
                }
            }
        }
    }
} else if (is.null(yinc)) {
    # Existing variable, no yinc, so update will be 0 or 1
    # Okay only if the current variable is 0/1
    if (all(is.na(newvar) | newvar==0L | newvar==1L))
        newvar[index!=0L] <- 1L
    else stop("tdc update does not match prior variable type: ", argname[ii])
} else {
    # attempt to update an existing tdc with new values from a new variable
    # We know how to handle a few special cases, but the basic strategy
    #  is "don't try to be clever".  Remember that class() can return
    #  a vector of length > 1
    if (inherits(newvar, "factor") && (!inherits(yinc, "factor") ||
                             !identical(levels(newvar), levels(yinc))))
        stop("tdc update does not match prior factor: ", argname[ii])
    clnew <- class(yinc)
    clold <- class(newvar)
```

```
        if (identical(clnew, clold) || (length(clnew)==1 && length(clold)==1  &&
                                 class(newvar) %in% c("integer", "numeric") &&
                                 class(yinc)   %in% c("integer", "numeric")))
            newvar[index != 0L] <- yinc[index]
        else stop("tdc update does not match prior variable type: ", argname[ii])
    }
    tdcvar <- unique(c(tdcvar, argname[[ii]]))
}
```

Events and cumevents are easy because each affects only one interval.

⟨*tmerge-addin2*⟩=

```
# add events
if (argclass[ii] %in% c("cumtdc", "cumevent")) {
    if (is.null(yinc)) yinc <- rep(1L, length(id))
    else if (is.logical(yinc)) yinc <- as.numeric(yinc)  # allow cumulative T/F
    if (!is.numeric(yinc)) stop("invalid increment for cumtdc or cumevent")
}
if (argclass[ii] == "cumevent"){
    ykeep <- (yinc !=0)  # ignore the addition of a censoring event
    yinc <- unlist(tapply(yinc, match(id, baseid), cumsum))
}

if (argclass[ii] %in% c("event", "cumevent")) {
    if (!is.null(newvar)) {
        if (!argname[ii] %in% tevent) {
            #warning(paste0("non-event variable '", argname[ii], "' replaced by an event variable
            newvar <- NULL
        }
        else if (!is.null(yinc)) {
            if (class(newvar) != class(yinc))
               stop("attempt to update an event variable with a different type")
            if (is.factor(newvar) && !all(levels(yinc) %in% levels(newvar)))
               stop("attemp to update an event variable and levels do not match")
        }
    }

    if (is.null(yinc)) yinc <- rep(1L, length(id))
    if (is.null(newvar)) {
        if (is.numeric(yinc)) newvar <- rep(0L, nrow(newdata))
        else if (is.factor(yinc))
            newvar <- factor(rep(levels(yinc)[1], nrow(newdata)),
                             levels(yinc))
        else if (is.character(yinc)) newvar <- rep('', nrow(newdata))
        else if (is.logical(yinc)) newvar <- rep(FALSE, nrow(newdata))
        else stop("invalid value for a status variable")
    }
    keep <- (subtype==1 | subtype==3) # all other events are thrown away
    if (argclass[ii] == "cumevent") keep <- (keep & ykeep)
    newvar[indx2[keep]] <- yinc[keep]

    # add this into our list of 'this is an event type variable'
    if (!(argname[ii] %in% tevent)) {
        tevent <- c(tevent, argname[[ii]])
        if (is.factor(yinc)) tcens <- c(tcens, list(levels(yinc)[1]))
        else if (is.logical(yinc)) tcens <- c(tcens, list(FALSE))
        else if (is.character(yinc)) tcens <- c(tcens, list(""))
        else if (is.integer(yinc))   tcens <- c(tcens, list(0L))
        else tcens <- c(tcens, list(0))
        names(tcens) <- tevent
    }
}

else if (argclass[ii] == "cumtdc") {  # process a cumtdc variable
    # I don't have a good way to catch the reverse of this user error
    if (argname[[ii]] %in% tevent)
        stop("attempt to turn event variable", argname[[ii]], "into a cumtdc")

    keep <- itype != 2  # changes after the last interval are ignored
    indx <- ifelse(subtype==1, indx1,
                   ifelse(subtype==3, indx2+1L, indx2))

    # we want to pass the right kind of NA to the C code
    default <- argi$default
    if (is.null(default)) default <- as.numeric(topt$tdcstart)
    else {
        if (length(default) != 1) stop("tdc initial value must be of length 1")
        if (!is.numeric(default)) stop("cumtdc initial value must be numeric")
    }
    if (is.null(newvar)) {  # not overwriting a prior value
        if (is.null(argi$value)) newvar <- rep(0.0, nrow(newdata))
        else newvar <- rep(default, nrow(newdata))
    }

    # the increment must be numeric
    if (!is.numeric(newvar))
        stop("data and starting value do not agree on data type")
    # id can be any data type; feed integers to the C routine
    storage.mode(yinc) <- storage.mode(dstart) <- "double"
    storage.mode(newvar) <- storage.mode(etime) <- "double"
    newvar <- .Call(Ctmerge, match(baseid, baseid), dstart, newvar,
                    match(id, baseid)[keep], etime[keep],
                    yinc[keep], indx[keep])
}

newdata[[argname[ii]]] <- newvar
```

Finish up by adding the attributes and the class

⟨*tmerge-finish*⟩=
```
tm.retain <- list(tname = topt[c("idname", "tstartname", "tstopname")],
                  n= nrow(newdata))
if (length(tevent))
    tm.retain$tevent <- list(name = tevent, censor=tcens)
if (length(tdcvar)>0) tm.retain$tdcvar <- tdcvar
attr(newdata, "tm.retain") <- tm.retain
attr(newdata, "tcount") <- rbind(attr(data1, "tcount"), tcount)
attr(newdata, "call") <- Call

row.names(newdata) <- NULL  #These are a mess; kill them off.
# Not that it works: R just assigns new row names.
class(newdata) <- c("tmerge", "data.frame")
newdata
```

The summary routine is for checking: it simply prints out the attributes.

⟨*tmerge-print*⟩=
```
summary.tmerge <- function(object, ...) {
    if (!is.null(cl <- attr(object, "call"))) {
        cat("Call:\n")
        dput(cl)
        cat("\n")
    }

    print(attr(object, "tcount"))
}

# This could be smarter: if you only drop variables that are not known
# to tmerge then it would be okay.  But I currently like the "touch it
#  and it dies" philosophy
"[.tmerge" <- function(x, ..., drop=TRUE){
    class(x) <- "data.frame"
    attr(x, "tm.retain") <- NULL
    attr(x, "tcount") <- NULL
    attr(x, "call") <- NULL
    NextMethod(x)
    }
```

14 Linear models and contrasts

The primary contrast function is `yates`. This function does both simple and population contrasts; the name is a nod to the “Yates weighted means” method, the first population contrast that I know of. A second reason for the name is that the word “contrast” is already overused in the S/R lexicon. Both `yates` and `cmatrix` can be used with any model that returns the necessary portions, e.g., lm, coxph, or glm. They were written because I became embroiled in the “type III” controversy, and made it a goal to figure out what exactly it is that SAS does. If I had known that that quest would take multiple years would perhaps have never started.

Population contrasts can result in some head scratching. It is easy to create the predicted value for any hypothethical subject from a model. A population prediction holds some data values constant and lets the others range over a population, giving a mean predicted value or population average. Population predictions for two treatments are the familiar g-estimates of causal models. We can take sums or differences of these predictions as well, e.g. to ask if they are significantly different. What can’t be done is to work backwards from one of these contrasts to the populations, at least for continuous variables. If someone asks for an x contrast of 15-5 is this a sum of two population estimates at 15 and -5, or a difference? It’s always hard to guess the mind of a user. Therefore what is needed is a fitted model, the term (covariate) of interest, levels of that covariate, a desired comparison, and a population.

First is cmatrix routine. This is called by users to create a contrast matrix for a model, users can also construct their own contrast matrices. The result has two parts: the definition of a set of predicted values and a set of contrasts between those values. The routine requires a fit and a formula. The formula is simply a way to get a set of variable names: all those variables are the fixed ones in the population contrast, and all others form the “population”. The result will be a matrix or list that has a label attribute containing the name of the term; this is used in printouts in the obvious way. Suppose that our model was `coxph(Surv(time, status) age*sex + ph.ecog)`. Someone might want the population matrix for age, sex, ph.ecog, or age+sex. For the last it doesn’t matter if they say age+sex, age*sex, or age:sex.

```
⟨yates⟩=
 cmatrix <- function(fit, term,
                     test =c("global", "trend", "pairwise", "mean"),
                     levels, assign) {
     # Make sure that "fit" is present and isn't missing any parts.
     if (missing(fit)) stop("a fit argument is required")
     Terms <- try(terms(fit), silent=TRUE)

     if (inherits(Terms, "try-error"))
         stop("the fit does not have a terms structure")
     else Terms <- delete.response(Terms)   # y is not needed
     Tatt <- attributes(Terms)
     # a flaw in delete.response: it doesn't subset dataClasses
     Tatt$dataClasses <- Tatt$dataClasses[row.names(Tatt$factors)]
     test <- match.arg(test)

     if (missing(term)) stop("a term argument is required")
     if (is.character(term)) term <- formula(paste("~", term))
     else if (is.numeric(term)) {
         if (all(term == floor(term) & term >0 & term < length(Tatt$term.labels)))
             term <- formula(paste("~",
                                   paste(Tatt$term.labels[term], collapse='+')))
         else stop("a numeric term must be an integer between 1 and max terms in the fit")
         }
     else if (!inherits(term, "formula"))
         stop("the term must be a formula or integer")
     fterm <- delete.response(terms(term))
     fatt <- attributes(fterm)
     user.name <- fatt$term.labels  # what the user called it
     termname <- all.vars(fatt$variables)
     indx <- match(termname, all.vars(Tatt$variables))
     if (any(is.na(indx)))
         stop("variable ", termname[is.na(indx)], " not found in the formula")

     # What kind of term is being tested?  It can be categorical, continuous,
     #  an interaction of only categorical terms, interaction of only continuous
     #  terms, or a mixed interaction.
     # Key is a trick to get "zed" from ns(zed, df= dfvar)
     key <- sapply(Tatt$variables[-1], function(x) all.vars(x)[1])
     parts <- names(Tatt$dataClasses)[match(termname, key)]
     types <- Tatt$dataClasses[parts]
     iscat <- as.integer(types=="factor" | types=="character")
     if (length(iscat)==1) termtype <- iscat
     else  termtype <- 2 + any(iscat) + all(iscat)

     # Were levels specified?  If so we either simply accept them (continuous),
     #  or double check them (categorical)
     if (missing(levels)) {
         temp <- fit$xlevels[match(parts, names(fit$xlevels), nomatch=0)]
         if (length(temp) < length(parts))
             stop("continuous variables require the levels argument")
         levels <- do.call(expand.grid, c(temp, stringsAsFactors=FALSE))
     }
     else {  #user supplied
         if (is.list(levels)) {
             if (is.null(names(levels))) {
                 if (length(termname)==1) names(levels)== termname
                 else stop("levels list requires named elements")
             }
         }
         if (is.data.frame(levels) || is.list(levels)) {
             index1 <- match(termname, names(levels), nomatch=0)
             # Grab the cols from levels that are needed (we allow it to have
             #  extra, unused columns)
             levels <- as.list(levels[index1])
             # now, levels = the set of ones that the user supplied (which might
             #   be none, if names were wrong)
             if (length(levels) < length(termname)) {
                 # add on the ones we don’t have, using fit$xlevels as defaults
                 temp <- fit$xlevels[parts[index1==0]]
                 if (length(temp) > 0) {
                     names(temp) <- termname[index1 ==0]
                     levels <- c(levels, temp)
                 }
             }
             index2 <- match(termname, names(levels), nomatch=0)
             if (any(index2==0))
                 stop("levels information not found for: ", termname[index2==0])
             levels <- expand.grid(levels[index2], stringsAsFactors=FALSE)
             if (any(duplicated(levels))) stop("levels data frame has duplicates")
         }
         else if (is.matrix(levels)) {
             if (ncol(levels) != length(parts))
                 stop("levels matrix has the wrong number of columns")
             if (!is.null(dimnames(levels)[[2]])) {
                 index <- match(termname, dimnames(levels)[[2]], nomatch=0)
                 if (index==0)
                     stop("matrix column names do no match the variable list")
                 else levels <- levels[,index, drop=FALSE]
             } else if (ncol(levels) > 1)
                 stop("multicolumn levels matrix requires column names")
             if (any(duplicated(levels)))
                 stop("levels matrix has duplicated rows")
             levels <- data.frame(levels, stringsAsFactors=FALSE)
             names(levels) <- termname
          }
         else if (length(parts) > 1)
             stop("levels should be a data frame or matrix")
         else {
             levels <- data.frame(x=unique(levels), stringsAsFactors=FALSE)
             names(levels) <- termname
         }
     }

     # check that any categorical levels are legal
     for (i in which(iscat==1)) {
         xlev <- fit$xlevels[[parts[i]]]
         if (is.null(xlev))
             stop("xlevels attribute not found for", termname[i])
         temp <- match(levels[[i]], xlev)
         if (any(is.na(temp)))
             stop("invalid level for term", termname[i])
     }

     rval <- list(levels=levels, termname=termname)
     # Now add the contrast matrix between the levels, if needed
     if (test=="global") {
          ⟨cmatrix-build-default⟩
     }
     else if (test=="pairwise") {
          ⟨cmatrix-build-pairwise⟩
     }
     else if (test=="mean") {
          ⟨cmatrix-build-mean⟩
     }
     else {
          ⟨cmatrix-build-linear⟩
     }
     # the user can say "age" when the model has "ns(age)", but we need
     #   the more formal label going forward
     rval <- list(levels=levels, termname=parts, cmat=cmat, iscat=iscat)
     class(rval) <- "cmatrix"
     rval
 }
```

The default contrast matrix is a simple test of equality if there is only one term. If the term is the interaction of multiple categorical variables then we do an anova type decomposition. In other cases we currently fail.

```
⟨cmatrix-build-default⟩=
if (TRUE) {
#if (length(parts) ==1) {
    cmat <- diag(nrow(levels))
    cmat[, nrow(cmat)] <- -1   # all equal to the last
    cmat <- cmat[-nrow(cmat),, drop=FALSE]
}
else if (termtype== 4) { # anova type
    stop("not yet done 1")
}
else stop("not yet done 2")
```

The pairwise option creates a set of contrast matrices for all pairs of a factor.

```
⟨cmatrix-build-pairwise⟩=
nlev <- nrow(levels)  # this is the number of groups being compared
if (nlev < 2) stop("pairwise tests need at least 2 groups")
```

```
npair <- nlev*(nlev-1)/2
if (npair==1) cmat <- matrix(c(1, -1), nrow=1)
else {
    cmat <- vector("list", npair)
    k <- 1
    cname <- rep("", npair)
    for (i in 1:(nlev-1)) {
        temp <- double(nlev)
        temp[i] <- 1
        for (j in (i+1):nlev) {
            temp[j] <- -1
            cmat[[k]] <- matrix(temp, nrow=1)
            temp[j] <- 0
            cname[k] <- paste(i, "vs", j)
            k <- k+1
        }
    }
    names(cmat) <- cname
}
```

The mean option compares each to the overall mean.

⟨*cmatrix-build-mean*⟩=

```
ntest <- nrow(levels)
cmat <- vector("list", ntest)
for (k in 1:ntest) {
    temp <- rep(-1/ntest, ntest)
    temp[k] <- (ntest-1)/ntest
    cmat[[k]] <- matrix(temp, nrow=1)
}
names(cmat) <- paste(1:ntest, "vs mean")
```

The `linear` option is of interest for terms that have more than one column; the two most common cases are a factor variable or a spline. It forms a pair of tests, one for the linear and one for the nonlinear part. For non-linear functions such as splines we need some notion of the range of the data, since we want to be linear over the entire range.

⟨*cmatrix-build-linear*⟩=

```
cmat <- vector("list", 2)
cmat[[1]] <- matrix(1:ntest, 1, ntest)
cmat[[2]] <- diag(ntest)
attr(cmat, "nested") <- TRUE
if (is.null(levels[[1]])) {
    # a continuous variable, and the user didn't give levels for the test
    #  look up the call and use the knots
    tcall <- Tatt$predvars[[indx + 1]]  # skip the 'call'
    if (tcall[[1]] == as.name("pspline")) {
        bb <- tcall[["Boundary.knots"]]
        levels[[1]] <- seq(bb[1], bb[2], length=ntest)
    }
    else if (tcall[[1]] %in% c("ns", "bs")) {
        bb <- c(tcall[["Boundary.knots"]], tcall[["knots"]])
        levels[[1]] <- sort(bb)
    }
    else stop("don't know how to do a linear contrast for this term")
}
```

Here are some helper routines. Formulas are from chapter 5 of Searle. The sums of squares only makes sense within a linear model.

```
⟨yates⟩=
 gsolve <- function(mat, y, eps=sqrt(.Machine$double.eps)) {
     # solve using a generalized inverse
     # this is very similar to the ginv function of MASS
     temp <- svd(mat, nv=0)
     dpos <- (temp$d > max(temp$d[1]*eps, 0))
     dd <- ifelse(dpos, 1/temp$d, 0)
     # all the parentheses save a tiny bit of time if y is a vector
     if (all(dpos)) x <- drop(temp$u %*% (dd*(t(temp$u) %*% y)))
     else if (!any(dpos)) x <- drop(temp$y %*% (0*y)) # extremely rare
     else x <-drop(temp$u[,dpos] %*%(dd[dpos] * (t(temp$u[,dpos, drop=FALSE]) %*% y)))
     attr(x, "df") <- sum(dpos)
     x
 }

 qform <- function(var, beta) { # quadratic form b' (V-inverse) b
     temp <- gsolve(var, beta)
     list(test= sum(beta * temp), df=attr(temp, "df"))
 }
```

The next functions do the work. Some bookkeeping is needed for a missing value in beta: we leave that coefficient out of the linear predictor. If there are missing coefs then the variance matrix will not have those columns in any case. The nafun function asks if a linear combination is NA. It treats 0*NA as 0.

```
⟨yates⟩=
 estfun <- function(cmat, beta, varmat) {
     nabeta <- is.na(beta)
     if (any(nabeta)) {
         k <- which(!nabeta)  #columns to keep
         estimate <- drop(cmat[,k] %*% beta[k])  # vector of predictions
         evar <- cmat[,k] %*% varmat %*% t(cmat[,k, drop=FALSE])
         list(estimate = estimate, var=evar)
     }
```

```
    else {
        list(estimate = drop(cmat %*% beta),
             var = cmat %*% varmat %*% t(cmat))
    }
}

testfun <- function(cmat, beta, varmat, sigma2) {
    nabeta <- is.na(beta)
    if (any(nabeta)) {
        k <- which(!nabeta)  #columns to keep
        estimate <- drop(cmat[,k] %*% beta[k])  # vector of predictions
        temp <- qform(cmat[,k] %*% varmat %*% t(cmat[,k,drop=FALSE]), estimate)
        rval <- c(chisq=temp$test, df=temp$df)
    }
    else {
       estimate <- drop(cmat %*% beta)
       temp <- qform(cmat %*% varmat %*% t(cmat), estimate)
       rval <- c(chisq=temp$test, df=temp$df)
       }
    if (!is.null(sigma2)) rval <- c(rval, ss= unname(rval[1]) * sigma2)
    rval
}

nafun <- function(cmat, est) {
    used <- apply(cmat, 2, function(x) any(x != 0))
    any(used & is.na(est))
    }
```

Now for the primary function. The user may have a list of tests, or a single term. The first part of the function does the usual of grabbing arguments and then checking them. The fit object has to have the standard stuff: terms, assign, xlevels and contrasts. Attributes of the terms are used often enough that we copy them to `Tatt` to save typing. We will almost certainly need the model frame and/or model matrix as well.

In the discussion below I use x1 to refer to the covariates/terms that are the target, e.g. `test='Mask'` to get the mean population values for each level of the Mask variable in the solder data set, and x2 to refer to all the other terms in the model, the ones that we average over. These are also referred to as U and V in the vignette.

⟨*yates*⟩=
```
yates <- function(fit, term, population=c("data", "factorial", "sas"),
                  levels, test =c("global", "trend", "pairwise"),
                  predict="linear", options, nsim=200,
                  method=c("direct", "sgtt")) {
    Call <- match.call()
    if (missing(fit)) stop("a fit argument is required")
    Terms <- try(terms(fit), silent=TRUE)
```

```
 if (inherits(Terms, "try-error"))
     stop("the fit does not have a terms structure")
 else Terms <- delete.response(Terms)   # y is not needed
 Tatt <- attributes(Terms)
 # a flaw in delete.response: it doesn't subset dataClasses
 Tatt$dataClasses <- Tatt$dataClasses[row.names(Tatt$factors)]

 if (inherits(fit, "coxphms")) stop("multi-state coxph not yet supported")
 if (is.list(predict) || is.function(predict)) {
     # someone supplied their own
     stop("user written prediction functions are not yet supported")
 }
 else {  # call the method
     indx <- match(c("fit", "predict", "options"), names(Call), nomatch=0)
     temp <- Call[c(1, indx)]
     temp[[1]] <- quote(yates_setup)
     mfun <- eval(temp, parent.frame())
 }
 if (is.null(mfun)) predict <- "linear"

# we will need the original model frame and X matrix
 mframe <- fit$model
 if (is.null(mframe)) mframe <- model.frame(fit)
 Xold <- model.matrix(fit)
 if (is.null(fit$assign)) { # glm models don't save assign
     xassign <- attr(Xold, "assign")
 }
 else xassign <- fit$assign


 nvar <- length(xassign)
 nterm <- length(Tatt$term.names)
 termname <- rownames(Tatt$factors)
 iscat <- sapply(Tatt$dataClasses,
                 function(x) x %in% c("character", "factor"))

 method <- match.arg(casefold(method), c("direct", "sgtt")) #allow SGTT
 if (method=="sgtt" && missing(population)) population <- "sas"

 if (inherits(population, "data.frame")) popframe <- TRUE
 else if (is.character(population)) {
     popframe <- FALSE
     population <- match.arg(tolower(population[1]),
                             c("data", "factorial", "sas",
                               "empirical", "yates"))
     if (population=="empirical") population <- "data"
```

```
    if (population=="yates") population <- "factorial"
}
else stop("the population argument must be a data frame or character")
test <- match.arg(test)

if (popframe || population != "data") weight <- NULL
else {
    weight <- model.extract(mframe, "weights")
    if (is.null(weight)) {
        id <- model.extract(mframe, "id")
        if (!is.null(id)) { # each id gets the same weight
            count <- c(table(id))
            weight <- 1/count[match(id, names(count))]
        }
    }
}

if (method=="sgtt" && (population !="sas" || predict != "linear"))
    stop("sgtt method only applies if population = sas and predict = linear")

beta <-  coef(fit, complete=TRUE)
nabeta <- is.na(beta)  # undetermined coefficients
vmat <-  vcov(fit, complete=FALSE)
if (nrow(vmat) > sum(!nabeta)) {
    # a vcov method that does not obey the complete argument
    vmat <- vmat[!nabeta, !nabeta]
}

# grab the dispersion, needed for the writing an SS in linear models
if (class(fit)[1] =="lm") sigma <- summary(fit)$sigma
else sigma <- NULL   # don't compute an SS column

# process the term argument and check its legality
if (missing(levels))
    contr <- cmatrix(fit, term, test, assign= xassign)
else contr <- cmatrix(fit, term, test, assign= xassign, levels = levels)
x1data <- as.data.frame(contr$levels)  # labels for the PMM values

# Make the list of X matrices that drive everything: xmatlist
#  (Over 1/2 the work of the whole routine)
xmatlist <- yates_xmat(Terms, Tatt, contr, population, mframe, fit,
                            iscat)

# check rows of xmat for estimability
⟨yates-estim-setup⟩
```

```
    # Drop missing coefficients, and use xmatlist to compute the results
    beta <- beta[!nabeta]
    if (predict == "linear" || is.null(mfun)) {
        # population averages of the simple linear predictor
        ⟨yates-linear⟩
    }
    else {
        ⟨yates-nonlinear⟩
    }
    result$call <- Call
    class(result) <- "yates"
    result
}
```

Models with factor variables may often lead to population predictions that involve non-estimable functions, particularly if there are interactions and the user specifies a factorial population. If there are any missing coefficients we have to do formal checking for this: any given row of the new X matrix, for prediction, must be in the row space of the original X matrix. If this is true then a regression of a new row on the old X will have residuals of zero. It is not possible to derive this from the pattern of NA coefficients alone. Set up a function that returns a true/false vector of whether each row of a matrix is estimable. This test isn't relevant if population=none.

```
⟨yates-estim-setup⟩=
if (any(is.na(beta)) && (popframe || population != "none")) {
    Xu <- unique(Xold)  # we only need unique rows, saves time to do so
    if (inherits(fit, "coxph")) X.qr <- qr(t(cbind(1.0,Xu)))
    else  X.qr <- qr(t(Xu))   # QR decomposition of the row space
    estimcheck <- function(x, eps= sqrt(.Machine$double.eps)) {
        temp <- abs(qr.resid(X.qr, t(x)))
        # apply(abs(temp), 1, function(x) all(x < eps)) # each row estimable
        all(temp < eps)
    }
    estimable <- sapply(xmatlist, estimcheck)
} else estimable <- rep(TRUE, length(xmatlist))
```

When the prediction target is $X\beta$ there is a four step process: build the reference population, create the list of X matrices (one prediction matrix for each for x1 value), column means of each X form each row of the contrast matrix Cmat, and then use Cmat to get the pmm values and tests of the pmm values.

```
⟨yates-linear⟩=
#temp <- match(contr$termname, colnames(Tatt$factors))
#if (any(is.na(temp)))
#    stop("term ’", contr$termname[is.na(temp)], "’ not found in the model")

meanfun <- if (is.null(weight)) colMeans else function(x) {
    colSums(x*weight)/ sum(weight)}
```

```
Cmat <- t(sapply(xmatlist, meanfun))[,!nabeta]

# coxph model: the X matrix is built as though an intercept were there (the
#  baseline hazard plays that role), but then drop it from the coefficients
#  before computing estimates and tests.  If there was a strata * covariate
#  interaction there will be many more colums to drop.
if (inherits(fit, "coxph")) {
    nkeep <- length(fit$means)  # number of non-intercept columns
    col.to.keep <- seq(to=ncol(Cmat), length= nkeep)
    Cmat <- Cmat[,col.to.keep, drop=FALSE]
    offset <- -sum(fit$means[!nabeta] * beta)  # recenter the predictions too
    }
else offset <- 0

# Get the PMM estimates, but only for estimable ones
estimate <- cbind(x1data, pmm=NA, std=NA)
if (any(estimable)) {
    etemp <- estfun(Cmat[estimable,,drop=FALSE], beta, vmat)
    estimate$pmm[estimable] <- etemp$estimate + offset
    estimate$std[estimable] <- sqrt(diag(etemp$var))
}

# Now do tests on the PMM estimates, one by one
if (method=="sgtt") {
        ⟨yates-sgtt⟩
}
else {
    if (is.list(contr$cmat)) {
        test <- t(sapply(contr$cmat, function(x)
                         testfun(x %*% Cmat, beta, vmat, sigma^2)))
        natest <- sapply(contr$cmat, nafun, estimate$pmm)
    }
    else {
        test <- testfun(contr$cmat %*% Cmat, beta, vmat, sigma^2)
        test <- matrix(test, nrow=1,
                       dimnames=list("global", names(test)))
        natest <- nafun(contr$cmat, estimate$pmm)
    }
    if (any(natest)) test[natest,] <- NA
}
if (any(estimable)){
#    Cmat[!estimable,] <- NA
    result <- list(estimate=estimate, test=test, mvar=etemp$var, cmat=Cmat)
    }
else  result <- list(estimate=estimate, test=test, mvar=NA)
if (method=="sgtt") result$SAS <- Smat
```

In the non-linear case the mfun object is either a single function or a list containing two functions `predict` and `summary`. The predict function is handed a vector $\eta = X\beta$ along with the X matrix, though most methods don't use X. The result of predict can be a vector or a matrix. For coxph models we add on an "intercept coef" that will center the predictions.

⟨*yates-nonlinear*⟩=
```
xall <- do.call(rbind, xmatlist)[,!nabeta, drop=FALSE]
if (inherits(fit, "coxph")) {
    xall <- xall[,-1, drop=FALSE]  # remove the intercept
    eta <- xall %*% beta -sum(fit$means[!nabeta]* beta)
}
else eta <- xall %*% beta
n1 <- nrow(xmatlist[[1]])  # all of them are the same size
index <- rep(1:length(xmatlist), each = n1)
if (is.function(mfun)) predfun <- mfun
else {  # double check the object
    if (!is.list(mfun) ||
        any(is.na(match(c("predict", "summary"), names(mfun)))) ||
        !is.function(mfun$predic) || !is.function(mfun$summary))
        stop("the prediction should be a function, or a list with two functions")
    predfun <- mfun$predict
    sumfun  <- mfun$summary
}
pmm <- predfun(eta, xall)
n2 <- length(eta)
if (!(is.numeric(pmm)) || !(length(pmm)==n2 || nrow(pmm)==n2))
    stop("prediction function should return a vector or matrix")
pmm <- rowsum(pmm, index, reorder=FALSE)/n1
pmm[!estimable,] <- NA

# get a sample of coefficients, in order to create a variance
# this is lifted from the mvtnorm code (can't include a non-recommended
# package in the dependencies)
tol <- sqrt(.Machine$double.eps)
if (!isSymmetric(vmat, tol=tol, check.attributes=FALSE))
    stop("variance matrix of the coefficients is not symmetric")
ev <- eigen(vmat, symmetric=TRUE)
if (!all(ev$values >= -tol* abs(ev$values[1])))
    warning("variance matrix is numerically not positive definite")
Rmat <- t(ev$vectors %*% (t(ev$vectors) * sqrt(ev$values)))
bmat <- matrix(rnorm(nsim*ncol(vmat)), nrow=nsim) %*% Rmat
bmat <- bmat + rep(beta, each=nsim)  # add the mean

# Now use this matrix of noisy coefficients to get a set of predictions
# and use those to create a variance matrix
# Since if Cox we need to recenter each run
```

```
sims <- array(0., dim=c(nsim, nrow(pmm), ncol(pmm)))
if (inherits(fit, 'coxph')) offset <- bmat %*% fit$means[!nabeta]
else offset <- rep(0., nsim)

for (i in 1:nsim)
    sims[i,,] <- rowsum(predfun(xall %*% bmat[i,] - offset[i]), index,
                        reorder=FALSE)/n1
mvar <- var(sims[,,1])  # this will be used for the tests
estimate <- cbind(x1data, pmm=unname(pmm[,1]), std= sqrt(diag(mvar)))

# Now do the tests, on the first column of pmm only
if (is.list(contr$cmat)) {
    test <- t(sapply(contr$cmat, function(x)
        testfun(x, pmm[,1], mvar[estimable, estimable], NULL)))
    natest <- sapply(contr$cmat, nafun, pmm[,1])
}
else {
    test <- testfun(contr$cmat, pmm[,1], mvar[estimable, estimable], NULL)
    test <- matrix(test, nrow=1,
                   dimnames=list(contr$termname, names(test)))
    natest <- nafun(contr$cmat, pmm[,1])
}
if (any(natest)) test[natest,] <- NA
if (any(estimable))
    result <- list(estimate=estimate,test=test, mvar=mvar)
else  result <- list(estimate=estimate, test=test, mvar=NA)

# If there were multiple columns from predfun, compute the matrix of
#  results and variances
if (ncol(pmm) > 1 && any(estimable)){
    pmm <-  apply(sims, 2:3, mean)
    mvar2 <- apply(sims, 2:3, var)
    # Call the summary function, if present
    if (is.list(mfun)) result$summary <- sumfun(pmm, mvar2)
    else {
        result$pmm <- pmm
        result$mvar2 <- mvar2
    }
}
```

Build the population data set. If the user provided a data set as the population then the task is fairly straightforward: we manipulate the data set and then call model.frame followed by model.matrix in the usual way. The primary task in that case is to verify that the data has all the needed variables.

Otherwise we have to be subtle.

1. We have ready access to a model frame, but not to the data. Consider a spline term for

instance — it's not always possible to go backwards and get the data.

2. We need to manipulate this model frame, e.g., make everyone treatment=A, then repeat with everyone treatment B.
3. We need to do it in a way that makes the frame still look like a correct model frame to R. This requires care.

For population= factorial we create a population data set that has all the combinations. If there are three adjusters z1, z2 and z3 with 2, 3, and 5 levels, respectively, the new data set will have 30 rows. If the primary model didn't have any z1*z2*z3 terms in it we likely could get by with less, but it's not worth the programming effort to figure that out: predicted values are normally fairly cheap. For population=sas we need a mixture: categoricals are factorial and others are data. Say there were categoricals with 3 and 5 levels, so the factorial data set has 15 obs, while the overall n is 50. We need a data set of 15*50 observations to ensure all combinations of the two categoricals with each continuous line.

An issue with data vs model is names. Suppose the original model was `lm(y ~ns(age,4) + factor(ph.ecog))`. In the data set the variable name is ph.ecog, in the model frame, the xlevels list, and terms structure it is factor(ph.ecog). The data frame has individual columns for the four variables, the model frame is a list with 3 elements, one of which is named "ns(age, 4)": notice the extra space before the 4 compared to what was typed.

```
⟨yates⟩=
 yates_xmat <- function(Terms, Tatt, contr, population, mframe, fit,
                        iscat, weight) {
     # which variables(s) are in x1 (variables of interest)
     # First a special case of strata(grp):x, which causes strata(grp) not to
     #  appear as a column
     if (any(is.na(match(contr$termname, colnames(Tatt$factors))))) {
         #tis rare
         if (length(contr$termname) > 1) stop("incomplete code 1")
         x1indx <- (contr$termname== rownames(Tatt$factors))
         names(x1indx) <- rownames(Tatt$factors)
         if (!any(x1indx)) stop(paste("variable", contr$termname, "not found"))
     } else x1indx <- apply(Tatt$factors[,contr$termname,drop=FALSE] >0, 1, any)
     x2indx <- !x1indx  # adjusters
     if (inherits(population, "data.frame")) pdata <- population  #user data
     else if (population=="data") pdata <- mframe  #easy case
     else if (population=="factorial")
         pdata <- yates_factorial_pop(mframe, Terms, x2indx, fit$xlevels)
     else if (population=="sas") {
         if (all(iscat[x2indx]))
             pdata <- yates_factorial_pop(mframe, Terms, x2indx, fit$xlevels)
         else if (!any(iscat[x2indx])) pdata <- mframe # no categoricals
         else { # mixed population
             pdata <- yates_factorial_pop(mframe, Terms, x2indx & iscat,
                                          fit$xlevels)
```

```
            n2 <- nrow(pdata)
            pdata <- pdata[rep(1:nrow(pdata), each=nrow(mframe)), ]
            row.names(pdata) <- 1:nrow(pdata)
            # fill in the continuous
            k <- rep(1:nrow(mframe), n2)
            for (i in which(x2indx & !iscat)) {
                j <- names(x1indx)[i]
                if (is.matrix(mframe[[j]]))
                    pdata[[j]] <- mframe[[j]][k,, drop=FALSE]
                else pdata[[j]] <- (mframe[[j]])[k]
                attributes(pdata[[j]]) <- attributes(mframe[[j]])
            }
        }
    }
    else stop("unknown population")  # this should have been caught earlier

    # Now create the x1 data set, the unique rows we want to test
    ⟨yates-x1mat⟩

    xmatlist
}
```

Build a factorial data set from a model frame.

```
⟨yates⟩=
yates_factorial_pop <- function(mframe, terms, x2indx, xlevels) {
    x2name <- names(x2indx)[x2indx]
    dclass <- attr(terms, "dataClasses")[x2name]
    if (!all(dclass %in% c("character", "factor")))
        stop("population=factorial only applies if all the adjusting terms are categorical")

    nvar <- length(x2name)
    n2 <- sapply(xlevels[x2name], length)  # number of levels for each
    n <- prod(n2)                          # total number of rows needed
    pdata <- mframe[rep(1, n), -1]  # toss the response
    row.names(pdata) <- NULL        # throw away funny names
    n1 <- 1
    for (i in 1:nvar) {
        j <- rep(rep(1:n2[i], each=n1), length=n)
        xx <- xlevels[[x2name[i]]]
        if (dclass[i] == "factor")
            pdata[[x2name[i]]] <- factor(j, 1:n2[i], labels= xx)
        else pdata[[x2name[i]]] <- xx[j]
        n1 <- n1 * n2[i]
    }
    attr(pdata, "terms") <- terms
    pdata
}
```

The next section builds a set of X matrices, one for each level of the x1 combination. The following was learned by reading the source code for model.matrix:

- If pdata has no terms attribute then model.matrix will call model.frame first, otherwise not. The xlev argument is passed forward to model.frame but is otherwise unused.
- If necessary, it will reorder the columns of pdata to match the terms, though I try to avoid that.
- Toss out the response variable, if present.
- Any character variables are turned into factors. The dataClass attribute of the terms object is not consulted.
- For each column that is a factor
  - if it alreay has a contrasts attribute, it is left alone.
  - otherwise a contrasts attribute is added using a matching element from contrasts.arg, if present, otherwise the global default
  - contrasts.arg must be a list, but it does not have to contain all factors
- Then call the internal C code

If pdata already is a model frame we want to leave it as one, so as to avoid recreating the raw data. If x1data comes from the user though, so we need to do that portion of model.frame processing ourselves, in order to get it into the right form. Always turn characters into factors, since individual elements of `xmatlist` will have only a subset of the x1 variables. One nuisance is name matching. Say the model had `factor(ph.ecog)` as a term; then `fit$xlevels` will have ‘factor(ph.ecog)’ as a name but the user will likely have created a data set using ‘ph.ecog’ as the name.

⟨*yates-x1mat*⟩=

```
if (is.null(contr$levels)) stop("levels are missing for this contrast")
x1data <- as.data.frame(contr$levels)  # in case it is a list
x1name <- names(x1indx)[x1indx]
for (i in 1:ncol(x1data)) {
    if (is.character(x1data[[i]])) {
        if (is.null(fit$xlevels[[x1name[i]]]))
            x1data[[i]] <- factor(x1data[[i]])
        else x1data[[i]] <- factor(x1data[[i]], fit$xlevels[[x1name[i]]])
    }
}

xmatlist <- vector("list", nrow(x1data))
if (is.null(attr(pdata, "terms"))) {
    np <- nrow(pdata)
    k <- match(x1name, names(pdata), nomatch=0)
    if (any(k>0)) pdata <- pdata[, -k, drop=FALSE]  # toss out yates var
    for (i in 1:nrow(x1data)) {
        j <- rep(i, np)
        tdata <- cbind(pdata, x1data[j,,drop=FALSE]) # new data set
        xmatlist[[i]] <- model.matrix(Terms, tdata, xlev=fit$xlevels,
                                      contrast.arg= fit$contrasts)
    }
} else {
    # pdata is a model frame, convert x1data
    # if the name and the class agree we go forward simply
    index <- match(names(x1data), names(pdata), nomatch=0)

    if (all(index >0) &&
        identical(lapply(x1data, class), lapply(pdata, class)[index]) &
        identical(sapply(x1data, ncol) , sapply(pdata, ncol)[index]))
            { # everything agrees
        for (i in 1:nrow(x1data)) {
            j <- rep(i, nrow(pdata))
            tdata <- pdata
            tdata[,names(x1data)] <- x1data[j,]
            xmatlist[[i]] <- model.matrix(Terms, tdata,
                                           contrasts.arg= fit$contrasts)
        }
    }
    else {
        # create a subset of the terms structure, for x1 only
        #  for instance the user had age=c(75, 75, 85) and the term was ns(age)
        # then call model.frame to fix it up
        x1term <- Terms[which(x1indx)]
        x1name <- names(x1indx)[x1indx]
        attr(x1term, "dataClasses") <- Tatt$dataClasses[x1name] # R bug
        x1frame <- model.frame(x1term, x1data, xlev=fit$xlevels[x1name])
        for (i in 1:nrow(x1data)) {
            j <- rep(i, nrow(pdata))
            tdata <- pdata
            tdata[,names(x1frame)] <- x1frame[j,]
            xmatlist[[i]] <- model.matrix(Terms, tdata, xlev=fit$xlevels,
                                      contrast.arg= fit$contrasts)
        }
    }
}
```

The decompostion based algorithm for SAS type 3 tests. Ignore the set of contrasts cmat since the algorithm can only do a global test. We mostly mimic the SAS GLM algorithm.

For the generalized Cholesky decomposition $LDL' = X'X$, where L is lower triangular with $L_{ii} = 1$ and D is diagonal, the set of contrasts $L'\beta$ gives the type I sequential sums of squares, partitioning the rows of L into those for term 1, term 2, etc. If X is the design matrix for a balanced factorial design then it is also true that $L_{ij} = 0$ unless term j includes term i, e.g., x1:x2 includes x1. These blocks of zeros mean that changing the order of the terms in the model simply rearranges L, and individual tests are unchanged.

This is precisely the definition of a type III contrast in SAS. With a bit of reading between the lines the "four types of estimable functions" document suggests the following algorithm:

1. Start with an X matrix in standard order of intercept, main effects, first order interactions, etc. Code any categorical variable with k levels as k 0/1 columns. An interaction of two categoricals with k and l levels will have kl columns, etc.

2. Create the dependency matrix $D = (X'X)^-(X'X)$. If column i of X can be written as a linear combination of prior columns, then column i of D contains that combination. Other columns of D match the identity matrix.

3. Intitialize $L = D$.

4. For any row i and j such that i is contained in j, make L_i orthagonal to L_j.

The algorithm appears to work in almost all cases, an exception is when the type 3 test has fewer degrees of freedom that we would expect.

Continuous variables are not orthagonalized in the SAS type III approach, nor any interaction that contains a continuous variable as one of its parts. To find the nested terms first note which rows of `factors` refer to categorical variables (the `iscat` variable); columns of `factors` that are non-zero only in categorical rows are the "categorical" columns. A term represented by one column in `factors` "contains" the term represented in some other column iff it's non-zero elements are a superset.

We have to build a new X matrix that is the expanded SAS coding, and are only able to do that for models that have an intercept, and use contr.treatement or contr.SAS coding.

⟨*yates-sgtt*⟩=

```
 # It would be simplest to have the contrasts.arg to be a list of function names.
 # However, model.matrix plays games with the calling sequence, and any function
 #  defined at this level will not be seen.  Instead create a list of contrast
 #  matrices.
 temp <- sapply(fit$contrasts, function(x) (is.character(x) &&
                            x %in% c("contr.SAS", "contr.treatment")))
 if (!all(temp))
         stop("yates sgtt method can only handle contr.SAS or contr.treatment")
 temp <- vector("list", length(fit$xlevels))
 names(temp) <- names(fit$xlevels)
 for (i in 1:length(fit$xlevels)) {
     cmat <- diag(length(fit$xlevels[[i]]))
     dimnames(cmat) <- list(fit$xlevels[[i]], fit$xlevels[[i]])
     if (i>1 || Tatt$intercept==1) {
         if (fit$contrasts[[i]] == "contr.treatment")
```

```
            cmat <- cmat[, c(2:ncol(cmat), 1)]
    }
    temp[[i]] <- cmat
}
sasX <- model.matrix(formula(fit),  data=mframe, xlev=fit$xlevels,
                      contrasts.arg=temp)
sas.assign <- attr(sasX, "assign")

# create the dependency matrix D.  The lm routine is unhappy if it thinks
#  the right hand and left hand sides are the same, fool it with I().
# We do this using the entire X matrix even though only categoricals will
#  eventually be used; if a continuous variable made it NA we need to know.
D <- coef(lm(sasX ~ I(sasX) -1))
dimnames(D)[[1]] <- dimnames(D)[[2]] #get rid if the I() names
zero <- is.na(D[,1])  # zero rows, we'll get rid of these later
D <- ifelse(is.na(D), 0, D)

# make each row orthagonal to rows for other terms that contain it
#  Containing blocks, if any, will always be below
# this is easiest to do with the transposed matrix
# Only do this if both row i and j are for a categorical variable
if (!all(iscat)) {
    # iscat marks variables in the model frame as categorical
    # tcat marks terms as categorical.  For x1 + x2 + x1:x2 iscat has
    # 2 entries and tcat has 3.
    tcat <- (colSums(Tatt$factors[!iscat,,drop=FALSE]) == 0)
}
else tcat <- rep(TRUE, max(sas.assign)) # all vars are categorical

B <- t(D)
dimnames(B)[[2]] <- paste0("L", 1:ncol(B))  # for the user
if (ncol(Tatt$factors) > 1) {
    share <- t(Tatt$factors) %*% Tatt$factors
    nc <- ncol(share)
    for (i in which(tcat[-nc])) {
        j <- which(share[i,] > 0 & tcat)
        k <- j[j>i]  # terms that I need to regress out
        if (length(k)) {
            indx1 <- which(sas.assign ==i)
            indx2 <- which(sas.assign %in% k)
            B[,indx1] <- resid(lm(B[,indx1] ~ B[,indx2]))
        }
    }
}

# Cut B back down to the non-missing coefs of the original fit
```

```
 Smat <- t(B)[!zero, !zero]
 Sassign <- xassign[!nabeta]
```

Although the SGTT does test for all terms, we only want to print out the ones that were asked for.

```
⟨yates-sgtt⟩=
 keep <- match(contr$termname, colnames(Tatt$factors))
 if (length(keep) > 1) { # more than 1 term in the model
     test <- t(sapply(keep, function(i)
                    testfun(Smat[Sassign==i,,drop=FALSE], beta, vmat, sigma^2)))
     rownames(test) <- contr$termname
 }  else {
     test <- testfun(Smat[Sassign==keep,, drop=FALSE], beta, vmat, sigma^2)
     test <- matrix(test, nrow=1,
                    dimnames=list(contr$termname, names(test)))
 }
```

The print routine places the population predicted values (PPV) alongside the tests on those values. Defaults are copied from printCoefmat.

```
⟨yates⟩=
 print.yates <- function(x, digits = max(3, getOption("digits") -2),
                         dig.tst = max(1, min(5, digits-1)),
                         eps=1e-8, ...) {
     temp1 <- x$estimate
     temp1$pmm <- format(temp1$pmm, digits=digits)
     temp1$std <- format(temp1$std, digits=digits)

     # the spaces help separate the two parts of the printout
     temp2 <- cbind(test= paste("    ", rownames(x$test)),
                    data.frame(x$test), stringsAsFactors=FALSE)
     row.names(temp2) <- NULL

     temp2$Pr <- format.pval(pchisq(temp2$chisq, temp2$df, lower.tail=FALSE),
                             eps=eps, digits=dig.tst)
     temp2$chisq <- format(temp2$chisq, digits= dig.tst)
     temp2$df <- format(temp2$df)
     if (!is.null(temp2$ss)) temp2$ss <- format(temp2$ss, digits=digits)

     if (nrow(temp1) > nrow(temp2)) {
         dummy <- temp2[1,]
         dummy[1,] <- ""
         temp2 <- rbind(temp2, dummy[rep(1, nrow(temp1)-nrow(temp2)),])
         }
     if (nrow(temp2) > nrow(temp1)) {
         # get rid of any factors before padding
```

```
        for (i in which(sapply(temp1, is.factor)))
            temp1[[i]] <- as.character(temp1[[i]])

        dummy <- temp1[1,]
        dummy[1,] <- ""
        temp1 <- rbind(temp1, dummy[rep(1, nrow(temp2)- nrow(temp1)),])
        }
    print(cbind(temp1, temp2), row.names=FALSE)
    invisible(x)
}
```

Routines to allow yates to interact with other models. Each is called with the fitted model and the type of prediction. It should return NULL when the type is a linear predictor, since the parent routine has a very efficient approach in that case. Otherwise it returns a function that will be applied to each value η, from each row of a prediction matrix.

⟨*yates*⟩=

```
yates_setup <- function(fit, ...)
    UseMethod("yates_setup", fit)

yates_setup.default <- function(fit, type, ...) {
    if (!missing(type) && !(type %in% c("linear", "link")))
        warning("no yates_setup method exists for a model of class ",
                class(fit)[1], " and estimate type ", type,
                ", linear predictor estimate used by default")
    NULL
}

yates_setup.glm <- function(fit, predict = c("link", "response", "terms",
                                          "linear"), ...) {
    type <- match.arg(predict)
    if (type == "link" || type== "linear") NULL # same as linear
    else if (type == "response") {
        finv <- family(fit)$linkinv
        function(eta, X) finv(eta)
    }
    else if (type == "terms")
        stop("type terms not yet supported")
}
```

For the coxph routine, we are making use of the R environment by first defining the baseline hazard and then defining the predict and summary functions. This means that those functions have access to the baseline.

⟨*yates*⟩=

```
yates_setup.coxph <- function(fit, predict = c("lp", "risk", "expected",
                                     "terms", "survival", "linear"),
                          options, ...) {
    type <- match.arg(predict)
    if (type=="lp" || type == "linear") NULL
    else if (type=="risk") function(eta, X) exp(eta)
    else if (type == "survival") {
        # If there are strata we need to do extra work
        # if there is an interaction we want to suppress a spurious warning
        suppressWarnings(baseline <- survfit(fit, censor=FALSE))
        if (missing(options) || is.null(options$rmean))
            rmean <- max(baseline$time)  # max death time
        else rmean <- options$rmean

        if (!is.null(baseline$strata))
            stop("stratified models not yet supported")
        cumhaz <- c(0, baseline$cumhaz)
        tt <- c(diff(c(0, pmin(rmean, baseline$time))), 0)

        predict <- function(eta, ...) {
            c2 <- outer(exp(drop(eta)), cumhaz)  # matrix of values
            surv <- exp(-c2)
            meansurv <- apply(rep(tt, each=nrow(c2)) * surv, 1, sum)
            cbind(meansurv, surv)
        }
        summary <- function(surv, var) {
            bsurv <- t(surv[,-1])
            std <- t(sqrt(var[,-1]))
            chaz <- -log(bsurv)
            zstat <- -qnorm((1-baseline$conf.int)/2)
            baseline$lower <- exp(-(chaz + zstat*std))
            baseline$upper <- exp(-(chaz - zstat*std))
            baseline$surv <- bsurv
            baseline$std.err  <- std/bsurv
            baselinecumhaz <- chaz
            baseline
        }
        list(predict=predict, summary=summary)
     }
    else stop("type expected is not supported")
}
```

15 The cox.zph function

The simplest test of proportional hazards is to use a time dependent coefficient $\beta(t) = a + bt$. Then $\beta(t)x = ax + b * (tx)$, and the extended coefficients a and b can be obtained from a Cox model with an extra 'fake' covariate tx. More generally, replace t with some function $g(t)$, which gives rise to an entire family of tests. An efficient assessment of this extended model can be done using a score test.

- Augment the original variables $x_1, \ldots x_k$ with k new ones $g(t)x_1, \ldots, g(t)x_k$
- Compute the first and second derivatives U and H of the Cox model at the starting estimate of $(\hat\beta, 0)$; prior covariates at their prior values, and the new covariates at 0. No iteration is done. This can be done efficiently with a modified version of the primary C routines for coxph.
- By design, the first k elements of U will be zero. Thus the first iteration of the new coefficients, and the score tests for them, are particularly easy.

The information or Hessian matrix for a Cox model is

$$\sum_{j \in deaths} V(t_j) = \sum_j V_j$$

where V_j is the variance matrix of the weighted covariate values, over all subjects at risk at time t_j. Then the expanded information matrix for the score test is

$$\begin{aligned}
H &= \begin{pmatrix} H_1 & H_2 \\ H_2' & H_3 \end{pmatrix} \\
H_1 &= \sum V(t_j) \\
H_2 &= \sum V(t_j) g(t_j) \\
H_3 &= \sum V(t_j) g^2(t_j)
\end{aligned}$$

The inverse of the matrix will be more numerically stable if $g(t)$ is centered at zero, and this does not change the test statistic. In the usual case $V(t)$ is close to constant in time — the variance of X does not change rapidly — and then H_2 is approximately zero. The original cox.zph used an approximation, which is to assume that $V(t)$ is exactly constant. In that case $H_2 = 0$ and $H_3 = \sum V(t_j) \sum g^2(t_j)$ and the test is particularly easy to compute. This assumption of identical components can fail badly for models with a covariate by strata interaction, and for some models with covariate dependent censoring. Multi-state models finally forced a change.

The newer version of the routine has two separate tracks: for the formal test and another for the residuals.

```
⟨cox.zph⟩=
 cox.zph <- function(fit, transform='km', terms=TRUE, singledf =FALSE,
                     global=TRUE) {
     Call <- match.call()
     if (!inherits(fit, "coxph") && !inherits(fit, "coxme"))
         stop ("argument must be the result of Cox model fit")
     if (inherits(fit, "coxph.null"))
         stop("there are no score residuals for a Null model")
```

```
    if (!is.null(attr(terms(fit), "specials")[["tt"]]))
        stop("function not defined for models with tt() terms")

    if (inherits(fit, "coxme")) {
        # drop all mention of the random effects, before getdata
        fit$formula <- fit$formula$fixed
        fit$call$formula <- fit$formula
     }

    cget <- coxph.getdata(fit, y=TRUE, x=TRUE, stratax=TRUE, weights=TRUE)
    y <- cget$y
    ny <- ncol(y)
    event <- (y[,ny] ==1)
    if (length(cget$strata))
        istrat <- as.integer(cget$strata) - 1L # number from 0 for C
    else istrat <- rep(0L, nrow(y))

    # if terms==FALSE the singledf argument is moot, but setting a value
    #   leads to a simpler path through the code
    if (!terms) singledf <- FALSE

    ⟨zph-setup⟩
    ⟨zph-transform⟩
    ⟨zph-terms⟩
    ⟨zph-schoen⟩

    rval$transform <- tname
    rval$call <- Call
    class(rval) <- "cox.zph"
    return(rval)
}

print.cox.zph <- function(x, digits = max(options()$digits - 4, 3),
                          signif.stars=FALSE, ...)  {
    invisible(printCoefmat(x$table, digits=digits, signif.stars=signif.stars,
                           P.values=TRUE, has.Pvalue=TRUE, ...))
}
```

The user can use t or $g(t)$ as the multiplier of the covariates. The default is to use the KM, only because that seems to be best at avoiding edge cases.

```
⟨zph-transform⟩=
times <- y[,ny-1]
if (is.character(transform)) {
    tname <- transform
    ttimes <- switch(transform,
                     'identity'= times,
                     'rank'    = rank(times),
                     'log'     = log(times),
                     'km' = {
                         temp <- survfitKM(factor(rep(1L, nrow(y))),
                                           y, se.fit=FALSE)
                         # A nuisance to do left continuous KM
                         indx <- findInterval(times, temp$time, left.open=TRUE)
                         1.0 - c(1, temp$surv)[indx+1]
                     },
                     stop("Unrecognized transform"))
        }
    else {
        tname <- deparse(substitute(transform))
        if (length(tname) >1) tname <- 'user'
        ttimes <- transform(times)
        }
    gtime <- ttimes - mean(ttimes[event])

    # Now get the U, information, and residuals
    if (ny==2) {
        ord <- order(istrat, y[,1]) -1L
        resid <- .Call(Czph1, gtime, y, X, eta,
                        cget$weights, istrat, fit$method=="efron", ord)
    }
    else {
        ord1 <- order(-istrat, -y[,1]) -1L   # reverse time for zph2
        ord  <- order(-istrat, -y[,2]) -1L
        resid <- .Call(Czph2, gtime, y, X, eta,
                        cget$weights, istrat, fit$method=="efron",
                        ord1, ord)
    }
```

The result has a score vector of length $2p$ where p is the number of variables and an information matrix that is $2p$ by $2p$. This is done with C code that is a simple variation on iteration 1 for a coxph model.

If singledf is TRUE then treat each term as a single degree of freedom test, otherwise as a multi-degree of freedom. If terms=FALSE test each covariate individually. If all the variables are univariate this is a moot point. The survival routines return Splus style assign components, that is a list with one element per term, each element an integer vector of coefficient indices.

The asgn vector is our main workhorse: loop over asgn to process term by term.

- if term=FALSE, set make a new asgn with one coef per term
- if a coefficient is NA, remove it from the relevant asgn vector
- frailties and penalized coxme coefficients are ignored: remove their element from the asgn list

For random effects models, including both frailty and coxme results, the random effect is included in the linear.predictors component of the fit. This allows us to do score tests for the other terms while effectively holding the random effect fixed.

If there are any NA coefficients these are redundant variables. It's easiest to simply get rid of them at the start by fixing up X, varnames, asgn, nvar, and fcoef. The variable matrix won't have the NA columns.

⟨*zph-setup*⟩=
```
eta <- fit$linear.predictors
X <- cget$x
varnames <- names(fit$coefficients)
nvar <- length(varnames)

if (!terms) {
    # create a fake asgn that has one value per coefficient
    asgn <- as.list(1:nvar)
    names(asgn) <- names(fit$coefficients)
}
else if (inherits(fit, "coxme")) {
    asgn <- attrassign(cget$x, terms(fit))
    # allow for a spelling inconsistency in coxme, later fixed
    if (is.null(fit$linear.predictors))
        eta <- fit$linear.predictor
    fit$df <- NULL  # don't confuse later code
}
else   asgn <- fit$assign

if (!is.list(asgn)) stop ("unexpected assign component")

frail <- grepl("frailty(", names(asgn), fixed=TRUE) |
         grepl("frailty.gamma(", names(asgn), fixed = TRUE) |
         grepl("frailty.gaussian(", names(asgn), fixed = TRUE)

if (any(frail)) {
    dcol <- unlist(asgn[frail])    # remove these columns from X
    X <- X[, -dcol, drop=FALSE]
    asgn <- asgn[!frail]
    # frailties don't appear in the varnames, so no change there
}
nterm <- length(asgn)
termname <- names(asgn)

fcoef <- fit$coefficients
if (any(is.na(fcoef))) {
    keep <- !is.na(fcoef)
    varnames <- varnames[keep]
    X <- X[,keep]
    fcoef <- fcoef[keep]

    # fix up assign
    new <- unname(unlist(asgn))[keep] # the ones to keep
    asgn <- sapply(asgn, function(x) {
        i <- match(x, new, nomatch=0)
        i[i>0]})
    asgn <- asgn[sapply(asgn, length)>0]  # drop any that were lost
    termname <- names(asgn)
    nterm <- length(asgn)   # asgn will be a list
    nvar <- length(new)
}
```

The zph1 and zph2 functions do not consider penalties, so we need to add those back in after the call. Nothing needs to be done wrt the first derivative: we already ignore the first ncoef elements of the returned first derivative (u) vector, which would have had a penalty. The second portion of u is for beta=0, and all of the penalties that currently are implemented have first derivative 0 at 0. For the second derivative, the current penalties (frailty, rigde, pspline) have a second derivative penalty that is independent of beta-hat. The coxph result contains the numeric value of the penalty at the solution, and we use a score test that would penalize the new time*pspline() term in the same way as the pspline term was penalized.

If no coefficients were missing then allvar will be 1:n, otherwise it will have holes.

⟨*zph-terms*⟩=

```
test <- double(nterm+1)
df   <- rep(1L, nterm+1)
u0 <- rep(0, nvar)
if (!is.null(fit$coxlist2)) { # there are penalized terms
    pmat <- matrix(0., 2*nvar, 2*nvar) # second derivative penalty
    pmat[1:nvar, 1:nvar] <- fit$coxlist2$second
    pmat[1:nvar + nvar, 1:nvar + nvar] <- fit$coxlist2$second
    imatr <- resid$imat + pmat
}
else imatr <- resid$imat

for (ii in 1:nterm) {
    jj <- asgn[[ii]]
    kk <- c(1:nvar, jj+nvar)
    imat <- imatr[kk, kk]
    u <- c(u0, resid$u[jj+nvar])
    if (singledf && length(jj) >1) {
        vv <- solve(imat)[-(1:nvar), -(1:nvar)]
        t1 <- sum(fcoef[jj] * resid$u[jj+nvar])
        test[ii] <- t1^2 * (fcoef[jj] %*% vv %*% fcoef[jj])
        df[ii] <- 1
    }
    else {
        test[ii] <- drop(solve(imat,u) %*% u)
        if (is.null(fit$df)) df[ii] <- length(jj)
        else df[ii] <- fit$df[ii]
    }
}

#Global test
if (global) {
    u <- c(u0, resid$u[-(1:nvar)])
    test[nterm+1] <- solve(imatr, u) %*% u
    if (is.null(fit$df))  df[nterm+1]   <- nvar
    else df[nterm+1] <- sum(fit$df)

    tbl <- cbind(test, df, pchisq(test, df, lower.tail=FALSE))
    dimnames(tbl) <- list(c(termname, "GLOBAL"), c("chisq", "df", "p"))
}
else {
    tbl <- cbind(test, df, pchisq(test, df, lower.tail=FALSE))[1:nterm,, drop=FALSE]
    dimnames(tbl) <- list(termname, c("chisq", "df", "p"))
}

# The x, y, residuals part is sorted by time within strata; this is
#  what the C routine zph1 and zph2 return
indx <- if (ny==2) ord +1 else rev(ord) +1  # return to 1 based subscripts
indx <- indx[event[indx]]                   # only keep the death times
rval <- list(table=tbl, x=unname(ttimes[indx]), time=unname(y[indx, ny-1]))
if (length(cget$strata)) rval$strata <- cget$strata[indx]
```

The matrix of scaled Schoenfeld residuals is created one stratum at a time. The ideal for the residual $r(t_i)$, contributed by an event for subject i at time t_i is to use $r_i V^{-1}(t_i)$, the inverse of the variance matrix of X at that time and for the relevant stratum. What is returned as `resid$imat` is $\sum_i V(t_i)$. One option would have been to return all the individual $\hat{V}_i$ matrices, but that falls over when the number at risk is too small and it cannot be inverted. Option 2 would be to use a per stratum averge of the V_i, but that falls flat for models with a large number of strata, a nested case-control model for instance. We take a different average that may not be the best, but seems to be good enough and doesn't seem to fail.

1. The `resid$used` matrix contains the number of deaths for each strata (row) that contributed to the sum for each variable (column). The value is either 0 or the number of events in the stratum, zero for those variables that are constant within the stratum. From this we can get the number of events that contributed to each element of the `imat` total. Dividing by this gives a per-element average `vmean`.

2. For a given stratum, some of the covariates may have been unused. For any of those set the scaled Schoenfeld residual to NA, and use the other rows/columns of the `vmean` matrix to scale the rest.

Now if some variable x_1 has a large variance at some time points and a small variance at others, or a large variance in one stratum and a small variance in another, the above smoothing won't catch that subtlety. However we expect such an issue to be rare. The common problem of strata*covariate interactions is the target of the above manipulations.

⟨*zph-schoen*⟩=

```
# Watch out for a particular edge case: there is a factor, and one of the
#   strata happens to not use one of its levels.  The element of resid$used will
#   be zero, but it really should not.
used <-resid$used
for (i in asgn) {
    if (length(i) > 1 && any(used[,i] ==0))
        used[,i] <- apply(used[,i,drop=FALSE], 1, max)
}

# Make the weight matrix
wtmat <- matrix(0, nvar, nvar)
for (i in 1:nrow(used))
    wtmat <- wtmat + outer(used[i,], used[i,], pmin)
# with strata*covariate interactions (multi-state models for instance) the
#  imatr matrix will be block diagonal.  Don't divide these off diagonal zeros
#  by a wtmat value of zero.
vmean <- imatr[1:nvar, 1:nvar, drop=FALSE]/ifelse(wtmat==0, 1, wtmat)

sresid <- resid$schoen
if (terms && any(sapply(asgn, length) > 1)) { # collase multi-column terms
    temp <- matrix(0, ncol(sresid), nterm)
    for (i in 1:nterm) {
        j <- asgn[[i]]
        if (length(j) ==1) temp[j, i] <- 1
        else temp[j, i] <- fcoef[j]
    }

    sresid <- sresid %*% temp
    vmean <- t(temp) %*% vmean %*% temp
    used <- used[, sapply(asgn, function(x) x[1]), drop=FALSE]
}

dimnames(sresid) <- list(signif(rval$time, 4), termname)

# for each stratum, rescale the Schoenfeld residuals in that stratum
sgrp <- rep(1:nrow(used), apply(used, 1, max))
for (i in 1:nrow(used)) {
    k <- which(used[i,] > 0)
    if (length(k) >0)  { # there might be no deaths in the stratum
        j <- which(sgrp==i)
        if (length(k) ==1) sresid[j,k] <- sresid[j,k]/vmean[k,k]
        else sresid[j, k] <- t(solve(vmean[k, k], t(sresid[j, k, drop=FALSE])))
        sresid[j, -k] <- NA
    }
}

# Add in beta-hat.  For a term with multiple columns we are testing zph for
#  the linear predictor X\beta, which always has a coefficient of 1
for (i in 1:nterm) {
    j <- asgn[[i]]
    if (length(j) ==1) sresid[,i] <- sresid[,i] + fcoef[j]
    else sresid[,i] <- sresid[,i] +1
}

rval$y <- sresid
rval$var <- solve(vmean)
```

⟨*cox.zph*⟩=

```
"[.cox.zph" <- function(x, ..., drop=FALSE) {
    i <- ..1
    if (!is.null(x$strata)) {
        y2 <- x$y[,i,drop=FALSE]
        ymiss <- apply(is.na(y2), 1, all)
        if (any(ymiss)) {
            # some deaths played no role in these coefficients
            #  due to a strata * covariate interaction, drop unneeded rows
            z<- list(table=x$table[i,,drop=FALSE], x=x$x[!ymiss],
                     time= x$time[!ymiss],
                     strata = x$strata[!ymiss],
                     y = y2[!ymiss,,drop=FALSE],
                     var=x$var[i,i, drop=FALSE],
                     transform=x$transform, call=x$call)
            }
        else z<- list(table=x$table[i,,drop=FALSE], x=x$x, time= x$time,
                      strata = x$strata,
                      y = y2,  var=x$var[i,i, drop=FALSE],
                      transform=x$transform, call=x$call)
    }
    else
        z<- list(table=x$table[i,,drop=FALSE], x=x$x, time= x$time,
                 y = x$y[,i,drop=FALSE],
                 var=x$var[i,i, drop=FALSE],
                 transform=x$transform, call=x$call)
    class(z) <- class(x)
    z
}
```