

Survival Package Functions

Terry Therneau

December 7, 2012

Contents

1	Introduction	1
2	Cox Models	2
2.1	Coxph	2
3	Exact partial likelihood	14
4	Cox models	23
4.1	Predicted survival	23
4.2	The predict method	45
4.3	Concordance	56
5	Expected Survival	69
6	Person years	77
7	Accelerated Failure Time models	85
7.1	Residuals	85
8	Competing risks	93
8.1	Printing and plotting	110
9	Plotting survival curves	119

1 Introduction

Let us change our 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 maintenance, I have found that whenever 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 remarkably 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. The noweb.R file contains functions that can tangle the code (extract a given R function), but the creation of the pdf still requires the noweb executable itself. I am working towards correcting that.

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.

```
<coxph>≡
tt <- function(x) x
coxph <- function(formula, data, weights, subset, na.action,
  init, control, ties= c("efron", "breslow", "exact"),
  singular.ok =TRUE, robust=FALSE,
  model=FALSE, x=FALSE, y=TRUE, tt, method=ties, ...) {

  ties <- match.arg(ties)
  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")
  temp <- Call[c(1,indx)] # only keep the arguments we wanted
  temp[[1]] <- as.name('model.frame') # change the function called

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

```

else terms(formula, special, data=data)
if (is.R()) m <- eval(temp, parent.frame())
else m <- eval(temp, sys.parent())

if (nrow(m) ==0) stop("No (non-missing) observations")
Terms <- terms(m)

## 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(...)

Y <- model.extract(m, "response")
if (!inherits(Y, "Surv")) stop("Response must be a survival object")
type <- attr(Y, "type")
if (type!='right' && type!='counting')
  stop(paste("Cox model doesn't support \"", type,
            "\" survival data", sep=''))
weights <- model.weights(m)
data.n <- nrow(Y) #remember this before any time transforms

strats <- attr(Terms, "specials")$strata
if (length(strats)) {
  stemp <- untangle.specials(Terms, 'strata', 1)
  if (length(stemp$terms) >0) #beware strata by covariate interactions
    Terms2 <- Terms[-stemp$terms] #not needed for model.matrix later
  else Terms2 <- Terms
  if (length(stemp$vars)==1) strata.keep <- m[[stemp$vars]]
  else strata.keep <- strata(m[,stemp$vars], shortlabel=TRUE)
  strats <- as.numeric(strata.keep)
}
else Terms2 <- Terms

timetrans <- attr(Terms, "specials")$tt
if (length(timetrans)) {
  (coxph-transform)
}

```

```

  <coxph-setup>
  <coxph-penal>
  <coxph-compute>
  <coxph-finish>
}

```

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..` 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 (missing(tt) || 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 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(strats, -Y[,1], Y[,2])
      #newstrat marks the first obs of each strata
      newstrat <- as.integer(c(1, 1*(diff(strats[sorted])!=0)))
    }
    if (storage.mode(Y) != "double") storage.mode(Y) <- "double"
    counts <- .Call("coxcount1", Y[sorted,],
                  as.integer(newstrat))
    tindex <- sorted[counts$index]
  }
  else {
    if (length(strats)==0) {
      sort.end <- order(-Y[,2], Y[,3])
      sort.start<- order(-Y[,1])
      newstrat <- c(1L, rep(0, nrow(Y) -1))
    }
    else {
      sort.end <- order(strats, -Y[,2], Y[,3])
      sort.start<- order(strats, -Y[,1])
      newstrat <- c(1L, as.integer(diff(strats[sort.end])!=0))
    }
    if (storage.mode(Y) != "double") storage.mode(Y) <- "double"
    counts <- .Call("coxcount2", Y,
                  as.integer(sort.start -1L),
                  as.integer(sort.end -1L),
                  as.integer(newstrat))
    tindex <- counts$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.

```

<coxph-transform>+≡
m <- m[tindex,]
Y <- Surv(rep(counts$time, counts$nrisk), counts$status)
type <- 'right' # new Y is right censored, even if the old was (start, stop]
strats <- factor(rep(1:length(counts$nrisk), counts$nrisk))

```

```

weights <- model.weights(m)
for (i in 1:ntrans)
  m[[timetrans$var[i]]] <- (tt[[i]]) (m[[timetrans$var[i]]], Y[,1], strats,
                                     weights)

```

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"
SEXP coxcount1(SEXP y2, SEXP strat2) {
  int ntime, nrow;
  int i, j, n;
  int stratstart=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"
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.

```

<coxph-setup>≡
offset <- model.offset(m)
if (is.null(offset) | all(offset==0)) offset <- rep(0., nrow(m))

cluster<- attr(Terms, "specials")$cluster
if (length(cluster)) {
  robust <- TRUE #flag to later compute a robust variance
  tempc <- untangle.specials(Terms2, 'cluster', 1:10)
  ord <- attr(Terms2, 'order')[tempc$terms]
  if (any(ord>1)) stop ("Cluster can not be used in an interaction")
  cluster <- strata(m[,tempc$vars], shortlabel=TRUE) #allow multiples
  Terms2 <- Terms2[-tempc$terms]
}
else {
  if (!missing(robust)) warning("The robust option is deprecated")
  else robust <- FALSE
}

attr(Terms2, 'intercept') <- 1 #baseline hazard is always present
X <- model.matrix(Terms2, m)
# Attributes of X need to be saved away before the X <- X[,-1] line removes the
# intercept, since subscripting removes some of them!
Xatt <- attributes(X)
if (is.R()) {
  assign <- lapply(attrassign(X, Terms2)[-1], function(x) x-1)
}

```

```

    xlevels <- .getXlevels(Terms2, m)
    contr.save <- attr(X, 'contrasts')
  }
else {
  assign <- lapply(attr(X, 'assign')[-1], function(x) x -1)
  xvars <- as.character(attr(Terms2, 'variables'))
  xvars <- xvars[-attr(Terms2, 'response')]
  if (length(xvars) >0) {
    xlevels <- lapply(m[xvars], levels)
    xlevels <- xlevels[!unlist(lapply(xlevels, is.null))]
    if(length(xlevels) == 0)
      xlevels <- NULL
  }
  else xlevels <- NULL
  contr.save <- attr(X, 'contrasts')
}

X <- X[,-1, drop=F] #remove the intercept column

if (missing(init)) init <- NULL

Check for penalized terms in the model, and set up infrastructure for the fitting routines to
deal with them.
<coxph-penal>≡
  pterms <- sapply(m, inherits, 'coxph.penalty')
  if (any(pterms)) {
    pattr <- lapply(m[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, strats, offset, init=init,
                       control,
                       weights=weights, method=method,
                       row.names(m), pcols, pattr, assign)
  }

<coxph-compute>≡
  else {
    if( method=="breslow" || method=="efron") {
      if (type== 'right') fitter <- get("coxph.fit")
      else fitter <- get("agreg.fit")
    }
    else if (method=='exact') {

```

```

        if (type=="right") fitter <- get("coxexact.fit")
        else fitter <- get("agexact.fit")
    }
else stop(paste ("Unknown method", method))

fit <- fitter(X, Y, strats, offset, init, control, weights=weights,
            method=method, row.names(m))
}

<coxph-finish>≡
if (is.character(fit)) {
  fit <- list(fail=fit)
  if (is.R()) class(fit) <- 'coxph'
  else oldClass(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) warning(msg)
    else stop(msg)
  }
  fit$n <- data.n
  fit$nevent <- sum(Y[,ncol(Y)])
  fit$terms <- Terms
  fit$assign <- assign
  if (is.R()) class(fit) <- fit$method
  else oldClass(fit) <- fit$method[1]
  if (robust) {
    fit$naive.var <- fit$var
    fit$method <- method
    # a little sneaky here: by calling resid before adding the
    # na.action method, I avoid having missings re-inserted
    # I also make sure that it doesn't have to reconstruct X and Y
    fit2 <- c(fit, list(x=X, y=Y, weights=weights))
    if (length(strats)) fit2$strata <- strats
    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$linear.predictors <- 0*fit2$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
}
na.action <- attr(m, "na.action")
if (length(na.action)) fit$na.action <- na.action
if (model) {
if (length(timetrans)) {
# Fix up the model frame -- still in the thinking stage
m[[".surv."]] <- Y
m[[".strata."]] <- strats
stop("Time transform + model frame: code incomplete")
}
fit$model <- m
}
if (x) {
Xatt$dim <- attr(X, 'dim')
Xatt$dimnames <- attr(X, 'dimnames')
Xatt$assign <- Xatt$assign[-1]
attributes(X) <- Xatt
fit$x <- X
if (length(strats)) {
if (length(timetrans)) fit$strata <- strats
else fit$strata <- strata.keep
}
}
if (y) fit$y <- Y
}

```

If any of the weights were not 1, save the results. Add names to the means component, which are occasionally useful to `survfit.coxph`. Other objects below are used when we need to recreate a model frame.

```
<coxph-finish>+≡
  if (!is.null(weights) && any(weights!=1)) fit$weights <- weights
  names(fit$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$call <- Call
  fit$method <- method
  fit
```

3 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 $i = 1, \dots, d$ in the n at risk. Then for the exact partial likelihood, the contribution at this time point is

$$L = \sum_{i=1}^d \log(r_i) - \log(D)$$

$$\frac{\partial L}{\partial \beta_j} = x_{ij} - \frac{\partial D}{\partial \beta_j} D^{-1}$$

$$\frac{\partial^2 L}{\partial \beta_j \partial \beta_k} = (1/D^2) \left[D \frac{\partial^2 D}{\partial \beta_j \partial \beta_k} - \frac{\partial D}{\partial \beta_j} \frac{\partial D}{\partial \beta_k} \right]$$

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

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

where $S(d, n)$ is the set of all possible subsets of size d from n objects, and s_1, s_2, \dots 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, /ldot, 345.

The brute force computation of all subsets can take a very long time. Gail et al [1] show simple recursion formulas that speed this up considerably. Let $D(d, n)$ be the denominator with

d deaths and n subjects. Then

$$D(d, n) = r_n D(d-1, n-1) + D(d, n-1) \quad (1)$$

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

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

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 first dimension of the array. The initial condition $D(0, n) = 1$ is important to all three routines.

$\langle \text{excox-recur} \rangle \equiv$

```
double coxd0(int d, int n, double *score, double *dmat,
             int dmax) {
    double *dn;

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

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

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

$\langle \text{excox-recur} \rangle + \equiv$

```
double coxd1(int d, int n, double *score, double *dmat, double *d1,
             double *covar, int dmax) {
    int indx;

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

```

        coxd1(d-1, n-1, score, dmat, d1, covar, dmax);
    }
    return(d1[indx]);
}

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

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

```

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

```

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

<excox-recur>

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

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

```

```

int    sstart; /* starting obs of current strata */
double *score;
double *oldbeta;
double zbeta;
double newlk=0;
double temp;
int    halving; /*are we doing step halving at the moment? */
int    nrisk; /* 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 */
      maxdeath, /* max tied deaths within a strata */
      ndeath; /* number of deaths at the current time point */
double dtime; /* time value under current examination */
double *dmem0, **dmem1, *dmem2; /* pointers to memory */
double *dtemp; /* used for zeroing the memory */
double *d1; /* current first derivatives from coxd1 */
double d0; /* global sum from coxc0 */

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

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

<excox-setup>
<excox-strata>
<excox-iter0>
<excox-iter>
}

```

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

```

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

/*
** Set up the ragged array pointer to the X matrix,
** and pointers to time and status

```

```

*/
covar= dmatrix(REAL(covar2), nused, nvar);
time = REAL(y2);
status = time +nused;
strata = INTEGER(PROTECT(duplicate(strata2)));
offset = REAL(offset2);

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

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

```

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

```

<excox-strata>≡
strata[0] =1; /* in case the parent forgot */
dsize = 0;

maxdeath =0;
j=0; /* start of the strata */
for (i=0; i<nused;) {
  if (strata[i]==1) { /* first obs of a new strata */
    if (i>0) {

```

```

        /* 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 >dsize) dsize = maxdeath*nrisk;
    }
    maxdeath =0; /* max tied deaths at any time in this strata */
    nrisk=0;
    ndeath =0;
}
dtime = time[i];
ndeath =0; /*number tied here */
while (time[i] ==dtime) {
    nrisk++;
    ndeath += status[i];
    i++;
    if (i>=nused || strata[i] >0) break; /*tied deaths don't cross strata */
}
if (ndeath > maxdeath) maxdeath=ndeath;
}
if (maxdeath*nrisk >dsize) dsize = maxdeath*nrisk;
if (maxdeath >1) strata[j] = maxdeath;

/* Now allocate memory for the scratch arrays
   Each per-variable slice is of size dsize
*/
dmemt0t = dsize * ((nvar*(nvar+1))/2 + nvar + 1);
dmem0 = (double *) R_alloc(dmemt0t, 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 loglikelihood, first, and second derivatives at this time point are

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

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

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

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

```

<excox-addup>≡
for (i=0; i<nused; ) {
  if (strata[i] >0) { /* first obs of a new strata */
    maxdeath= strata[i];
    dtemp = dmem0;
    for (j=0; j<dmemtot; j++) *dtemp++ =0.0;
    sstart =i;
    nrisk =0;
  }

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

  /* We have added up over the death time, now process it */
  if (ndeath >0) { /* Add to the loglik */
    d0 = coxd0(ndeath, nrisk, score+sstart, dmem0, maxdeath);
    R_CheckUserInterrupt();
    newlk -= log(d0);
    dmem2 = dmem0 + (nvar+1)*dsize; /*start for the second deriv memory */
    for (j=0; j<nvar; j++) { /* for each covariate */
      d1[j] = coxd1(ndeath, nrisk, score+sstart, dmem0, dmem1[j],
        covar[j]+sstart, maxdeath) / d0;
      if (ndeath > 3) R_CheckUserInterrupt();
      u[j] -= d1[j];
      for (k=0; k<= j; k++) { /* second derivative*/
        temp = coxd2(ndeath, nrisk, score+sstart, dmem0, dmem1[j],
          dmem1[k], dmem2, covar[j] + sstart,
          covar[k] + sstart, maxdeath);
        if (ndeath > 5) R_CheckUserInterrupt();
        imat[k][j] += temp/d0 - d1[j]*d1[k];
        dmem2 += dsize;
      }
    }
  }
}

```

```

    }
}

```

Do the first iteration of the solution. The first iteration is different in 3 ways: it is used to set the initial log-likelihood, to compute the score test, and we pay no attention to convergence criteria or diagnostics. (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-adddup>

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

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

loglik[2] =0; /* score test stored here */
for (i=0; i<nvar; i++)
    loglik[2] += u[i]*d1[i];

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

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

```

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

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

```

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

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

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

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

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

      for (i=0; i<nvar; i++) {

```

```

        oldbeta[i] = beta[i];
        beta[i] = beta[i] + u[i];
    }
} /* return for another iteration */

/*
** Ran out of iterations
*/
loglik[1] = newlk;
loglik[3] = 1000; /* signal no convergence */
<excox-finish>
    The common code for finishing. Invert the information matrix, copy it to be symmetric, and
    put together the output structure.
<excox-finish>≡
    loglik[4] = iter;
    chinv2(imat, nvar);
    for (i=1; i<nvar; i++)
        for (j=0; j<i; j++) imat[i][j] = imat[j][i];

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

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

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

```

4 Cox models

4.1 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

type computation type for the survival curve

vartype computation type for the variance

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

id replacement and extension for the individual argument

newstrata does the new data set contain strata, and I should obey it?

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

<survfit.coxph>≡

```
survfit.coxph <-  
  function(formula, newdata, se.fit=TRUE, conf.int=.95, individual=FALSE,  
           type, vartype,  
           conf.type=c("log", "log-log", "plain", "none"),  
           censor=TRUE, id, newstrata=missing(id),  
           na.action=na.pass, ...) {
```

```
  Call <- match.call()
```

```
  Call[[1]] <- as.name("survfit") #nicer output for the user
```

```
  object <- formula      #'formula' because it has to match survfit
```

```
  <survfit.coxph-setup>
```

```
  <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 routine is fairly pedestrian. 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 the `type` or `vartype` are missing we use the appropriate one for the method in the Cox model. That is, the `coxph` computation used for `method='exact'` is the same approximation used in the Kalbfleish-Prentice estimate, that for the Breslow method matches the Aalen survival estimate, and the Efron approximation the Efron survival estimate. The other two rows of labels in `temp1` are historical; we include them for backwards compatibility but they don't appear in the documentation.

```

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

  if (missing(type)) {
    # Use the appropriate one from the model
    temp1 <- c("exact", "breslow", "efron")
    survtype <- match(object$method, temp1)
  }
  else {
    temp1 <- c("kalbfleisch-prentice", "aalen", "efron",
              "kaplan-meier", "breslow", "fleming-harrington",
              "greenwood", "tsiatis", "exact")
    survtype <- match(match.arg(type, temp1), temp1)
    survtype <- c(1,2,3,1,2,3,1,2,3)[survtype]
  }
  if (missing(vartype)) {
    vartype <- survtype
  }
  else {
    temp2 <- c("greenwood", "aalen", "efron", "tsiatis")
    vartype <- match(match.arg(vartype, temp2), temp2)
    if (vartype==4) vartype<- 2
  }

  if (!se.fit) conf.type <- "none"
  else conf.type <- match.arg(conf.type)

```

I need to retrieve a copy of the original data. We always need the X matrix and y , both of which may be found in the data object. If the original call included either `strata`, `offset`, or `weights`, 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-setup>+≡
  if (is.null(object$y) || is.null(object[['x']])) ||

```

```

    !is.null(object$call$weights) ||
    !is.null(attr(object$terms, 'specials')$strata) ||
    !is.null(attr(object$terms, 'offset')) {

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

  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. For instance

  fit <- coxph(Surv(time,status) ~ age, data=lung)
  lung <- lung[1:100,]
  survfit(fit)

  <survfit.coxph-setup>+≡
  if (is.null(mf)) y <- object[['y']]
  else {
    y <- model.response(mf)
    y2 <- object[['y']]
    if (!is.null(y2) && any(as.matrix(y2) != as.matrix(y)))
      stop("Could not reconstruct the y vector")
  }

  if (is.null(object[['x']])) x <- model.matrix.coxph(object, data=mf)
  else x <- object[['x']]

  n <- nrow(y)
  if (n != object$n[1] || nrow(x) !=n)
    stop("Failed to reconstruct the original data set")

  if (is.null(mf)) wt <- rep(1., n)
  else {
    wt <- model.weights(mf)
    if (is.null(wt)) wt <- rep(1.0, n)
  }

  type <- attr(y, 'type')
  if (type != 'right' && type != 'counting')
    stop("Cannot handle \"", type, "\" type survival data")
  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(0,n)
  else id <- NULL

```

```

if (individual && missing(newdata)) {
  warning("the id and/or individual options only make sense with new data")
  individual <- FALSE
}

if (individual && type!= 'counting')
  stop("The individual option is only valid for start-stop data")
if (!missing(newstrata)) {
  if (!is.logical(newstrata)) stop("newstrata must be TRUE/FALSE")
  if (individual && !newstrata)
    stop("newstrata must be TRUE for the individual or id options")
}
else newstrata <- individual

if (is.null(mf)) offset <- 0
else {
  offset <- model.offset(mf)
  if (is.null(offset)) offset <- 0
}

Terms <- object$terms
temp <- untangle.specials(Terms, 'strata')
if (length(temp$terms)==0) strata <- rep(OL,n)
else {
  if (length(temp$vars) ==1) strata <- mf[[temp$vars]]
  else strata <- strata(mf[, temp$vars], shortlabel=TRUE)
}

```

A model with strata by covariate interactions forms a special case for the calculations, one that had not been recognized in versions of the code before 1/1/2011. (Results were wrong, sometimes bizzare.) For example `coxph(Surv(time, status) ~ age * strata(sex), data=lung`. Create a vector `stype` that contains 0 for ordinary variables, 1 for strata terms, and 2 for strata by covariate interactions; this will be used later.

```

<survfit.coxph-setup>+≡
if (!is.null(x) && ncol(x) >0) { #not a ~1 or ~offset(x) model
  tempf <-attr(Terms, "factors")[-attr(Terms, 'response'),,drop=F]
  stype <- ifelse(colSums(tempf[temp$terms,,drop=FALSE]) >0,
                 attr(Terms, "order"), 0)
}
else stype <- 0 #dummy value

```

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 compatible. A non-sparse frailty does not need special code and works out like any other variable.

We also remove the means from each column of the X matrix. 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. One particular special case (that gave me fits for a while) is when there are non-hierarchical models, for example $\sim \text{age} + \text{age}:\text{sex}$. 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. This is simply a bad model and it is not clear that there is any "correct" behavior in creating predicted survival curves.

```

(survfit.coxph-setup)+≡
  if (is.null(x) || ncol(x)==0) { # a model with ~1 on the right hand side
    # Give it a dummy x so the rest of the code goes through
    # (This case is really rare)
    x <- matrix(0., nrow=n)
    coef <- 0.0
    varmat <- matrix(0.0,1,1)
    risk <- rep(exp(offset- mean(offset)), length=n)
  }
  else {
    varmat <- object$var
    coef <- ifelse(is.na(object$coefficients), 0, object$coefficients)
    xcenter <- object$means
    if (is.null(object$frail)) {
      x <- scale(x, center=xcenter, scale=FALSE)
      risk <- c(exp(x%% coef + offset - mean(offset)))
    }
    else {
      keep <- !is.na(match(dimnames(x)[[2]], names(coef)))
      x <- x[,keep, drop=F]
      # varmat <- varmat[keep,keep] #coxph already has trimmed it
      risk <- exp(object$linear.predictor)
      x <- scale(x, center=xcenter, scale=FALSE)
    }
  }
}

```

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 individual 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 response 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, ...)` won't work here for two reasons. The first is the decision that for a stratified model, if `newdata` does not contain the strata variable we will produce curves for all the strata. The above would fail in that case with a “variable not found” message. (If I could go back in time this feature might disappear.) The second is that we might have an `id` statement in this call, which is another variable to be fetched. 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 original model formula. Any `cluster` terms can be deleted, If `individual` is false then the response variable can go. If `!newstrata` we can remove strata variables. The reason for this is so that the program won't ask for variables in the new data frame when we won't need them. We have to be careful in subscripting a terms object, since this will remove the `predvars` attribute (at least currently). Both clauses below are trial and error guesses based on a set of examples. The `delete.response` function works correctly, however.

```

<survfit.coxph-setup>+≡
  subterms <- function(x, i) {
    dataClasses <- attr(x, "dataClasses")
    predvars <- attr(x, "predvars")
    x <- x[i]
    if (!is.null(predvars))
      attr(x, "predvars") <- attr(x, "variables")
    if (!is.null(dataClasses)){
      temp <- dimnames(attr(x, 'factors'))[[1]]
      attr(x, "dataClasses") <- dataClasses[temp]
    }
    x
  }
temp <- untangle.specials(Terms, 'cluster')
if (length(temp$vars)) {
  ptemp <- attr(temp, "predvars")
  Terms <- subterms(Terms, -temp$terms)
  stype <- stype[-temp$terms]
}
if (missing(newdata)) {
  mf2 <- as.list(object$means) #create a dummy newdata
  names(mf2) <- names(object$coefficients)
}

```

```

mf2 <- as.data.frame(mf2)
}
else {
  if (!is.null(object$frail))
    stop("Newdata cannot be used when a model has sparse frailty terms")

  Terms2 <- Terms
  if (!individual) Terms2 <- delete.response(Terms)
  if (!newstrata && (any(stype>0)))
    Terms2 <- subterms(Terms2,stype==0) #remove strata and interactions
  <survfit.coxph-newdata2>
}

```

Now get the newdata model frame. Two things need to be cleared up first. If xlevels contains any variable that is no longer in the model then one gets a spurious warning message out of the `model.frame` function. This happens if there was a strata object removed using `subterms` above. Second, for backwards compatibility, 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.null(object$xlevels)) {
  myxlev <- object$xlevels[match(attr(Terms2, "term.labels"),
                                names(object$xlevels), nomatch=0)]

  if (length(myxlev)==0) myxlev <- NULL
}
else myxlev <- NULL

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))
}

```

Finally get my new model frame `mf2`. If all is particularly simple we can use a simple call. Otherwise get an abbreviated form of the original call that has only the calling function, `na.action`, and `id`. The calling function is always element 1, the others are found by name. Now manipulate it: add the formula, data and xlev components (the last might be `NULL`), and then change the name of the call. If the original call was `survfit(fit1, newdata=mydat, conf.int=.9)` the result is `model.frame(data= copy of newdat, formula=Terms2, xlev=myxlev)`.

There is one more issue to trip you up, which is that `model.frame` is special. It totally ignores any scoping arguments passed to it, e.g. the the second and third arguments to `eval`. Variables are first searched for in the data, and then in the environment of the formula. What to do?

- We don't dare modify the environment of the formula. 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.

- Assign the data argument. The code originally would retain the “newdata” part of the call, and then rename it to “data” in the tcall list. But then it would not be found if survfit were called from inside another function. The assignment happens in this context so we are ok.

```

<survfit.coxph-newdata2>+=
  if (newstrata && missid)
    mf2 <- model.frame(Terms2, data=newdata, na.action=na.action, xlev=myxlev)
  else {
    tcall <- Call[c(1, match(c('id', "na.action"), names(Call), nomatch=0))]
    tcall$data <- newdata
    tcall$formula <- Terms2
    if (!is.null(object$xlevels)) tcall$xlev <- myxlev
    tcall[[1]] <- as.name('model.frame')
    mf2 <- eval(tcall)
  }

```

Now, finally, extract the x2 matrix from the just-created frame. For individual=TRUE there must be a newdata argument, it must have any strata variables that were present in the original model, and the levels of those strata must exist in the original model.

```

<survfit.coxph-result>=
  if (newstrata) {
    temp <- untangle.specials(Terms2, 'strata')
    if (length(temp$vars) > 0) {
      strata2 <- strata(mf2[temp$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")
      Terms2 <- Terms2[-temp$terms]
    }
    else strata2 <- factor(rep(0, nrow(mf2)))
  }

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

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

```

```

x2 <- scale(x2, center=xcenter, scale=FALSE)

offset2 <- model.offset(mf2)
if (length(offset2) >0) offset2 <- offset2 - mean(offset)
else 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

newrisk <- exp(c(x2 %*% coef) + offset2)
result <- survfitcoxph.fit(y, x, wt, x2, risk, newrisk, strata,
                          se.fit, survtype, vartype, varmat,
                          id, y2, strata2)
}

```

The odd case is when there are strata by covariate interactions, but `individual=FALSE`. In this case the `x2` matrix is different for each strata, something that the `survfitcoxph.fit` subroutine is not set up to do. This case is also very, very rare; the package existed for 15+ years before I became aware there even was a problem. Thus we take a low tech solution: call the fit routine for every possible strata combination, and pick off the result we want from each. This requires a model frame that is a blend of the original data `mf` and the new data `mf2`. You can't just paste these together, as I discovered in an evening of failures; the `'terms'` component of a model frame needs to be present and be compatible or `model.matrix` will fail with one of several utterly confusing messages. Start with `tframe`, a subset of rows of `mf` containing each unique strata. Create `mf3` as `nrow(mf2)` identical copies of one row of `tframe`, then fill in new values for the `mf2` variables. We now have an `mf` "copy" with all the right data, and it has all the right variable names in the right order, which is necessary to successfully paste the original `mf` terms attribute onto it.

```

<survfit.coxph-result>+≡
else if (any(stype==2)){
  tframe <- mf[match(levels(strata), strata),]
  temp <- vector('list', nrow(tframe)) #number of strata
  for (i in 1:nrow(tframe)) {
    mf3 <- tframe[rep(i, nrow(mf2)),]
    mf3[names(mf2)] <- mf2
    attr(mf3, 'terms') <- attr(mf, 'terms')
    x2 <- model.matrix(Terms[stype!=1], data=mf3,
                      xlev=object$xlevels)[,-1,drop=FALSE]
    x2 <- scale(x2, center=xcenter, scale=FALSE)
    offset2 <- model.offset(mf3)
    if (is.null(offset2)) offset2 <-0
    newrisk <- c(exp(x2%*%coef) + offset2)
  }
}

```

```

zed<- survfitcoxph.fit(y, x, wt, x2, risk, newrisk, strata,
                      se.fit, survtype, vartype, varmat,
                      id=NULL, unlist=FALSE)

temp[[i]] <- zed[[i]]
}
tfun <- function(x, fun) as.vector(unlist(lapply(x,fun))) #no names
result <- list(n = tfun(temp, function(x) x$n),
             time= tfun(temp, function(x) x$time),
             n.risk= tfun(temp, function(x) x$n.risk),
             n.event= tfun(temp, function(x) x$n.event),
             n.censor=tfun(temp, function(x) x$n.censor),
             strata = sapply(temp, function(x) length(x$time)))
names(result$strata) <- levels(strata)
if (nrow(x2) >1) { #matrix result
  result$surv = t(matrix(tfun(temp, function(x) t(x$surv)),
                       nrow= nrow(x2)))
  if (se.fit) result$std.err = t(matrix(tfun(temp,function(x) t(x$std)),
                                       nrow= nrow(x2)))
}
else {
  result$surv = tfun(temp, function(x) x$surv)
  if (se.fit) result$std.err = tfun(temp, function(x) x$std.err)
}
}

```

The last two are the easy cases. First if there is no newdata argument, the centering means that we need to predict for $x_2=0$. The second the most common call to the routine.

(survfit.coxph-result)+≡

```

else {
  if (missing(newdata)) {
    x2 <- matrix(0.0, nrow=1, ncol=ncol(x))
    offset2 <- 0
  }
  else {
    offset2 <- model.offset(mf2)
    if (length(offset2) >0) offset2 <- offset2 - mean(offset)
    else offset2 <- 0
    x2 <- model.matrix(Terms2, mf2)[,-1, drop=FALSE] #no intercept
    x2 <- scale(x2, center=xcenter, scale=FALSE)
  }
}

newrisk <- exp(c(x2 %*% coef) + offset2)
result <- survfitcoxph.fit(y, x, wt, x2, risk, newrisk, strata,
                          se.fit, survtype, vartype, varmat,
                          id, y2, strata2)

if (newstrata) {

```

```

    }
  }
}

```

The `newstrata` argument is a recent addition. If someone used it, then they don't want the usual behavior of all strata for each variable combination. We need to pick off of the right bits

```

<newstrata-fixup>≡
  warning("newstrata argument under construction, value ignored")

```

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 <- rep(names(result$strata), result$strata)
      result$strata <- c(table(temp[keep]))
    }
    result <- lapply(result, kfun, keep)
  }

  if (se.fit) {
    zval <- qnorm(1- (1-conf.int)/2, 0,1)
    if (conf.type=='plain') {
      temp1 <- result$surv + zval* result$std.err * result$surv
      temp2 <- result$surv - zval* result$std.err * result$surv
      result <- c(result, list(upper=pmin(temp1,1), lower=pmax(temp2,0),
                               conf.type='plain', conf.int=conf.int))
    }
    if (conf.type=='log') {
      xx <- ifelse(result$surv==0,1,result$surv) #avoid some "log(0)" messages
      temp1 <- ifelse(result$surv==0, 0*result$std.err,
                      exp(log(xx) + zval* result$std.err))
      temp2 <- ifelse(result$surv==0, 0*result$std.err,
                      exp(log(xx) - zval* result$std.err))
      result <- c(result, list(upper=pmin(temp1,1), lower=temp2,
                               conf.type='log', conf.int=conf.int))
    }
    if (conf.type=='log-log') {
      who <- (result$surv==0 | result$surv==1) #special cases
      xx <- ifelse(who, .1,result$surv) #avoid some "log(0)" messages
    }
  }

```

```

    temp1 <- exp(-exp(log(-log(xx)) + zval*result$std.err/log(xx)))
    temp1 <- ifelse(who, result$surv + 0*result$std.err, temp1)
    temp2 <- exp(-exp(log(-log(xx)) - zval*result$std.err/log(xx)))
    temp2 <- ifelse(who, result$surv + 0*result$std.err, temp2)
    result <- c(result, list(upper=temp1, lower=temp2,
                           conf.type='log-log', conf.int=conf.int))
  }
}

result$call <- Call

# The "type" component is in the middle -- match history
indx <- match('surv', names(result))
result <- c(result[1:indx], type=attr(y, 'type'), result[-(1:indx)])
if (is.R()) class(result) <- c('survfit.cox', 'survfit')
else      oldClass(result) <- 'survfit.cox'
result

```

Now, we're ready to do the main computation. Before this revision (the one documented here using `noweb`) there were three C routines used in calculating survival after a Cox model

1. `agsurv1` creates a single curve, but for the most general case of a *covariate path*. It is used for time dependent covariates.
2. `agsurv2` creates a set of curves. These curves are for a fixed covariate set, although (start, stop] data is supported. If there were 3 strata in the fit and 4 covariate sets are given, the result will be 12 curves.
3. `agsurv3` is used to create population survival curves. The result is average survival curve (for 3 different definitions of 'average'). If there were 3 strata and 100 subjects, the first curve returned would be the average for those 100 individual curves in strata 1, the second for strata 2, and the third for strata 3.

In June 2010 the first two were re-written in (mostly) R, in the process of adding functionality and repairing some flaws in the computation of a weighted variance. In effect, the changes are similar to the rewrite of the `survfitKM` function a few years ago.

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.

First, compute the baseline survival curves for each strata. If the strata was a factor we want to leave it in the same order, otherwise sort it. 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 different recreation of the x and y data.

```

⟨survfitcoxph.fit⟩≡
  survfitcoxph.fit <- function(y, x, wt, x2, risk, newrisk, strata, se.fit,
                             survtype, vartype, varmat, id, y2, strata2,

```

```

                                unlist=TRUE) {
if (is.factor(strata)) ustrata <- levels(strata)
else                    ustrata <- sort(unique(strata))
nstrata <- length(ustrata)
survlist <- vector('list', nstrata)

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", "std.err")]
    else result[[1]][c("n", "time", "n.risk", "n.event", "n.censor",
                      "surv")]
  }
  else {
    temp <-list(n      = unlist(lapply(result, function(x) x$n)),
              time=   unlist(lapply(result, function(x) x$time)),
              n.risk= unlist(lapply(result, function(x) x$n.risk)),
              n.event= unlist(lapply(result, function(x) x$n.event)),
              n.censor=unlist(lapply(result, function(x) x$n.censor)),
              strata = sapply(result, function(x) length(x$time)))
    names(temp$strata) <- ustrata

    if ((missing(id) || is.null(id)) && nrow(x2)>1) {
      temp$urv <- t(matrix(unlist(lapply(result,
                                         function(x) t(x$urv))),
                          nrow= nrow(x2)))
      if (se.fit) temp$std.err = t(matrix(unlist(lapply(result,
                                                         function(x) t(x$std.err))),
                                         nrow= nrow(x2)))
    }
    else {
      temp$urv <- unlist(lapply(result, function(x) x$urv))
      if (se.fit) temp$std.err <- unlist(lapply(result,
                                                function(x) x$std.err))
    }
    temp
  }
}

```

```

    }
  }
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 component 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.

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 i th new risk score r_i . The variance has two parts, the first of which is $r_i^2 H_1$ where H_1 is returned from the `agsurv` routine, and the second is

$$H_2(t) = d'(t)Vd(t)$$

$$d(t) = \int_0^t [z - \bar{x}(s)]d\Lambda(s)$$

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

```

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

  if (is.matrix(x2) && nrow(x2) >1) { #more than 1 row in newdata
    fit$surv <- outer(surv, newrisk, '^')
    if (se.fit) {
      varh <- matrix(0., nrow=length(fit$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))*

```

```

        newrisk[i]^2
      }
      fit$std.err <- sqrt(varh)
    }
  }
else {
  fit$surv <- surv^newrisk
  if (se.fit) {
    dt <- outer(fit$cumhaz, c(x2)) - fit$xbar
    varh <- (cumsum(fit$varhaz) + rowSums((dt %*% varmat)* dt)) *
      newrisk^2
    fit$std.err <- sqrt(varh)
  }
}
fit
}

```

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

Now, if `id` is missing we can simply apply the `expand` function to each strata. For the case with `id` 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(id) || is.null(id))
    result <- lapply(survlst, expand, x2, varmat, se.fit)
  else {
    onecurve <- function(slst, x2, y2, strata2, newrisk, se.fit) {

```

```

ntarget <- nrow(x2) #number of different time intervals
surv <- vector('list', ntarget)
n.event <- n.risk <- n.censor <- varh1 <- varh2 <- time <- surv
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])

    if (survtype==1) surv[[i]] <- slist$surv[indx]^newrisk[i]
    else surv[[i]] <- slist$hazard[indx]*newrisk[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] *newrisk[i]^2
    varh2[[i]] <- rowSums((dt %*% varmat)* dt) * newrisk[i]^2
  }
}

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

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,
       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)
    }

    if (all(id ==id[1])) {
      result <- list(onecurve(survlist, x2, y2, strata2, newrisk, se.fit))
    }
    else {
      uid <- unique(id)
      result <- vector('list', length=length(uid))
      for (i in 1:length(uid)) {
        indx <- which(id==uid[i])
        result[[i]] <- onecurve(survlist, x2[indx,,drop=FALSE],
                                y2[indx,,drop=FALSE],
                                strata2[indx], newrisk[indx], se.fit)
      }
      names(result) <- unique(id)
    }
  }
}

```

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 survival (`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 `who <- (start<t & stop>=t)` is $O(n)$ and can take more time than 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 initial sums. The problem here is potential round off error if the sums get large, which can happen if the time scale were very, very finely divided. This issue is mostly precluded by subtracting means first.

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

`<agsurv>`≡

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

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

```

    }

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

```

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

```

<agsurv>+≡
dtimes <- which(nevent >0)
ntime <- length(time)
if (survtype ==1) {
  indx <- (which(status==1))[order(dtime[status==1])] #deaths
  km <- .C('agsurv4',
    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) {
  xsum2 <- rowsum((wrisk*death) *x, dtime)
  erisk <- rowsum(wrisk*death, dtime) #risk score sums at each death
  tsum <- .C('agsurv5',
    as.integer(length(nevent)),
    as.integer(nvar),
    as.integer(ndeath),
    as.double(nrisk),
    as.double(erisk),
    as.double(xsum),
    as.double(xsum2),
    sum1 = double(length(nevent)),
    sum2 = double(length(nevent)),
    xbar = matrix(0., length(nevent), nvar))
}
haz <- switch(survtype,
  nevent/nrisk,
  nevent/nrisk,
  nevent* tsum$sum1)
varhaz <- switch(vartype,
  nevent/(nrisk *
    ifelse(nevent>=nrisk, nrisk, nrisk-nevent)),
  nevent/nrisk^2,
  nevent* tsum$sum2)
xbar <- switch(vartype,
  (xsum/nrisk)*haz,

```

```

        (xsum/nrisk)*haz,
        nevent * tsum$xbar)

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

```

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

```

<agsurv4>≡
#include "survS.h"
#include "survproto.h"

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

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

```

```

    }
    j += ndeath[i];
  }
}

```

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

n number of observations (unique death times)

d number of deaths at that time

nvar number of covariates

x1 weighted number at risk at the time

x2 sum of weights for the deaths

xsum matrix containing the cumulative sum of x values

xsum2 matrix of sums, only for the deaths

On output the values are

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

- d=1

sum1 $1/x1$

sum2 $1/x1^2$

xbar $xsum/x1^2$

- d=2

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

sum2 $(1/2) (\text{same terms, squared})$

xbar $(1/2) (xsum/x1^2 + (xsum - 1/2 x3)/(x1- x2/2)^2)$

- d=3

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

sum2 $(1/3) (\text{same terms, squared})$

xbar $(1/3) (xsum/x1^2 + (xsum - 1/3 xsum2)/(x1- x2/3)^2 + (xsum - 2/3 xsum2)/(x1- 2/3 x3)^2)$

- etc

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

```

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

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

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

```

4.2 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
- `terms` = a matrix with one row per subject and one column for each term in the model.

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

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

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

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

reference the reference context for centering the results

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

The first task of the routine is to reconstruct 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 original 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 specified `y=FALSE` in the original 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"),
                           se.fit=FALSE, na.action=na.pass,
                           terms=names(object$assign), collapse,
                           reference=c("strata", "sample"), ...) {
  <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.

```

<pcoxph-init>≡
  if (!inherits(object, 'coxph'))
    stop("Primary argument much be a coxph object")

  Call <- match.call()
  type <- match.arg(type)
  n <- object$n
  Terms <- object$terms

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

  # I will never need the cluster argument, if present delete it.
  # Terms2 are terms I need for the newdata (if present), y is only
  # needed there if type == 'expected'
  if (length(attr(Terms, 'specials')$cluster)) {
    temp <- untangle.specials(Terms, 'cluster', 1)
    Terms <- object$terms[-temp$terms]
  }
  else Terms <- object$terms

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

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

```

```
n <- length(object$residuals)
```

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

The next task of the routine is to reconstruct 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 original 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 specified `y=FALSE` in the original call. We also need the strata in this case. Grabbing it is the same amount of work as grabbing `X`, so gets lumped into the case below 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 <- model.frame(object)  
      y <- model.extract(mf, 'response')  
      have.mf <- TRUE #for the logic a few lines below, avoid double work  
    }  
  }  
  
if (se.fit || type=='terms' || (!missing(newdata) && type=="expected") ||  
    (has.strata && (reference=="strata") || type=="expected")) {  
  use.x <- TRUE  
  if (is.null(object[['x']]) || has.weights || has.offset ||  
      (has.strata && is.null(object$strata))) {  
    # I need the original model frame  
    if (!have.mf) mf <- model.frame(object)  
    if (nrow(mf) != n)
```

```

stop("Data is not the same size as it was in the original fit")

if (has.strata) {
  if (!is.null(object$strata)) oldstrat <- object$strata
  else {
    stemp <- untangle.specials(Terms, 'strata')
    if (length(stemp$vars)==1) oldstrat <- mf[[stemp$vars]]
    else oldstrat <- strata(mf[,stemp$vars], shortlabel=TRUE)
    # A model with x:strata(grp) will have stemp$terms of length 0
    if (length(stemp$terms))
      x <- model.matrix(Terms[-stemp$terms], mf,
                       contr=object$contrasts)[,-1,drop=FALSE]
    else x<- model.matrix(Terms, mf,
                          contr=object$contrasts)[,-1,drop=FALSE]
  }
}
else {
  x <- model.matrix(Terms, mf,
                   contr=object$contrasts)[,-1,drop=FALSE]
  oldstrat <- rep(0L, n)
}
weights <- model.weights(mf)
if (is.null(weights)) weights <- rep(1.0, n)
offset <- model.offset(mf)
if (is.null(offset)) offset <- 0
}
else {
  x <- object[['x']]
  if (has.strata) oldstrat <- object$strata
  else oldstrat <- rep(0L, n)
  weights <- rep(1.,n)
  offset <- 0
}
}
else {
  # I won't need strata in this case either
  if (has.strata) {
    stemp <- untangle.specials(Terms, 'strata', 1)
    Terms2 <- Terms2[-stemp$terms]
    has.strata <- FALSE #remaining routine never needs to look
  }
  oldstrat <- rep(0L, n)
  offset <- 0
  use.x <- FALSE
}
}

```

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

```

<pcoph-getdata>+=
  if (!missing(newdata)) {
    use.x <- TRUE #we do use an X matrix later
    tcall <- Call[c(1, match("newdata", 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[[1]] <- as.name('model.frame') # change the function called
    tcall$xlev <- object$xlevels
    if (is.R()) mf2 <- eval(tcall, parent.frame())
    else      mf2 <- eval(tcall, sys.parent())

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

```

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>
  }

```

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 computation 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))
  }

  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$xbar)[j1+1,,drop=F]
  }
  if (ncol(y)==2) {
    if (se.fit) {
      dt <- (chaz * newx[indx2,]) - xbar[indx2,]
      se[indx2] <- sqrt(varh + rowSums((dt %%% object$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[indx2,]
  dt2 <- (chaz2 * newx[indx2,]) - xbar2[indx2,]

  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

$$\eta_i = X_i\beta - (1/n) \sum_j X_j\beta \quad (7)$$

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 covariate 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 $(x - \bar{x})'\beta$ 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

`<pcoph-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) newx <- x - rep(object$means, each=nrow(x))
}
else {
  offset <- newoffset - mean(offset)
  if (has.strata && reference=="strata") {
    xmeans <- rowsum(x*weights, oldstrat)/c(rowsum(weights, oldstrat))
    newx <- newx - xmeans[match(newstrat, row.names(xmeans)),]
  }
  else newx <- newx - rep(object$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 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 $\text{var}(\text{sum}) = \text{sum}(\text{variances})$. 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)) {
  if (length(collapse) != n) 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

```

4.3 Concordance

The concordance statistic is gaining popularity as a measure of goodness-of-fit in survival models. Consider all pairs of subjects with (r_i, r_j) as the two risk scores for each pair and (s_i, s_j) the corresponding survival times. The c-statistic is defined by dividing these sets into four groups.

- Concordant pairs: for a Cox model this will be pairs where a shorter survival is paired with a larger risk score, e.g. $r_i > r_j$ and $s_i < s_j$
- Discordant pairs: the lower risk score has a shorter survival
- Tied pairs: there are three common choices
 - Kendall's tau: any pair where $r_i = r_j$ or $s_i = s_j$ is considered tied.
 - AUC: pairs with $r_i = r_j$ are tied; those with $s_i = s_j$ are considered incomparable. This is the definition of the AUC in logistic regression, and has become the most common choice for Cox models as well.
 - Somer's D: All ties are treated as incomparable.
- Incomparable pairs: For survival this always includes pairs where the survival times cannot be ranked with certainty. For instance s_i is censored at time 10 and s_j is an event (or censor) at time 20. Subject i may or may not survive longer than subject j . Note that if s_i is censored at time 10 and s_j is an event at time 10 then $s_i > s_j$. Add onto this those ties that are treated as incomparable. Observations that are in different strata are also incomparable, since the Cox model only compares within strata.

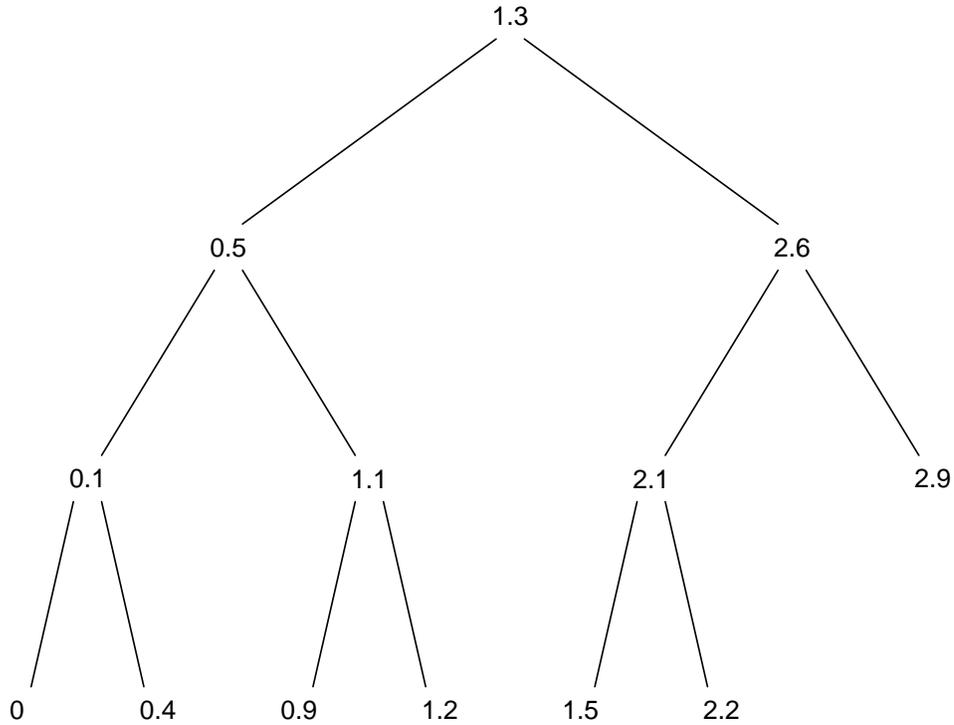


Figure 1: A balanced tree of 13 nodes.

Then the concordance statistic is defined as $(C + T/2)/(C + D + T)$. The denominator is the number of comparable pairs.

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 AUC definition, but all 4 values are reported back so that a user can recreate the others if desired.

The primary computational question is how to do this efficiently, i.e., better than the naive $O(n^2)$ 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 also neatly solves the question of time-dependent covariates.
2. To count the number higher and lower we need to rank the subjects in the risk set by their scores r_i . This can be done in $O(\log n)$ time if the 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 descendants 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 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 2.1. The concordant pairs are all of those with a risk score greater than 2.1, which can be found by traversing the tree from the top down, adding the (parent - child) value each time we branch left (5-3 at the 2.6 node), with a last addition of the right hand child when we find the node with Smith’s value (1). There are 3 concordant and $12-3=9$ 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. (In the code below we start at Smith’s node and walk up.)

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 and node counts are changed to zero as observations are removed.

If we 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 and the children are $2i + 1$ and $2i + 2$ and the parent is $(i - 1)/2$. The following bit of code returns the indices of a sorted list when placed into such a tree. The basic idea is that the rows of the tree start at indices 1, 2, 4, ... For the above tree, the last row will contains the 1st, 3rd, ..., 11th smallest ranks. The next row above contains every other value of the ranks *not yet assigned*, and etc to the top of the tree. There is some care to make sure the result is an integer.

```

<btree>≡
  btree <- function(n) {
    ranks <- rep(0L, n) #will be overwritten
    yet.to.do <- 1:n
    depth <- floor(logb(n,2))
    start <- as.integer(2^depth)
    lastrow.length <- 1+n-start
    indx <- seq(1L, by=2L, length= lastrow.length)
    ranks[yet.to.do[indx]] <- start + 0:(length(indx)-1L)
    yet.to.do <- yet.to.do[-indx]

    while (start >1) {
      start <- as.integer(start/2)
      indx <- seq(1L, by=2L, length=start)
      ranks[yet.to.do[indx]] <- start + 0:(start-1L)
      yet.to.do <- yet.to.do[-indx]
    }
    ranks
  }

```

Referring again to figure 1, `btree(13)` yields the vector 8 4 9 2 10 5 11 1 12 6 13 3 7 meaning that the smallest element will be in position 8 of the tree, the next smallest in position 4, etc.

Here is a shorter recursive version. It knows the form of trees with 1, 2, or 3 nodes; and builds the others from them. The maximum depth of recursion is $\log_2(n) - 1$. It is more clever but a bit slower. (Not that it matters as both take less than 5 seconds for a million elements.)

```

<btree2>≡
btree <- function(n) {
  tfun <- function(n, id, power) {
    if (n==1) id
    else if (n==2) c(2L *id, id)
    else if (n==3) c(2L*id, id, 2L*id +1L)
    else {
      nleft <- if (n== power*2) power else min(power-1, n-power/2)
      c(tfun(nleft, 2L *id, power/2), id,
        tfun(n-(nleft+1), 2L*id +1L, power/2))
    }
  }
  tfun(n, 1L, 2^(floor(logb(n-1,2))))
}

```

A second question is how to compute the variance of the result. The insight used here is to consider a Cox model with time dependent covariates, where the covariate x at each death time has been transformed into $\text{rank}(x)$. It is easy to show that the Cox score statistic contribution at each death is $(D - C)/2$ where C and D are the number of concordant and discordant pairs contributed at that death time (for a Cox fit using the Breslow approximation). The contribution to the variance of the score statistic is $V(t) = \sum (r_i - \bar{r})^2/n$, the r_i being the ranks at that time point and n the number at risk. How can we update this sum using an update formula? First remember the identity

$$\sum w_i(x_i - \bar{x})^2 = \sum w_i(x_i - c)^2 - \sum w_i(c - \bar{x})^2$$

true for any set of values x and centering constant c . For weighted data define the rank of an observation with risk score r_k as

$$\text{rank} = \sum_{r_i < r_k} w_i + (1/2) \sum_{r_i = r_k} w_i$$

These correspond to the midpoints of the rise on an empirical CDF, and for unweighted data without ties gives ranks of $.5, 1.5, \dots, n - .5$.

Assume we have just added obs k to the tree. Since the mean rank $= \mu_g = \sum(w_i)/2$ the grand mean increases by $w_k/2$. Let μ_ℓ be the mean rank for all observations currently in the tree of rank lower than r_k , the item we are about to add, μ_u be the mean for all those above in rank (after the addition), μ_g the grand mean, and μ_n the new grand mean after adding in subject k . We have

$$\begin{aligned} \mu_\ell &= \sum_{r_i < r_k} w_i/2 \\ \mu_u &= \sum_{r_i <= r_k} w_i + \sum_{r_i \geq r_k} w_i/2 \end{aligned}$$

For items of lower rank than r_k , none of their ranks will change with the addition of this new observation. This leads to the update formula on the third line below. (I'm using $i < k$ as shorthand for $r_i < r_k$ below)

$$\begin{aligned} \sum_{i < k} w_i(r_i - \mu_g)^2 &= \sum_{i < k} w_i(r_i - \mu_\ell)^2 + \left(\sum_{i < k} w_i\right)(\mu_\ell - \mu_g)^2 \\ \sum_{i < k} w_i(r_i - \mu_n)^2 &= \sum_{i < k} w_i(r_i - \mu_\ell)^2 + \left(\sum_{i < k} w_i\right)(\mu_\ell - \mu_n)^2 \\ \sum_{i < k} w_i(r_i - \mu_n)^2 - \sum_{i < k} w_i(r_i - \mu_g)^2 &= \left(\sum_{i < k} w_i\right)[(\mu_\ell - \mu_n)^2 - (\mu_\ell - \mu_g)^2] \\ &= \left(\sum_{i < k} w_i\right)(\mu_n + \mu_g - 2\mu_\ell)(\mu_n - \mu_g) \end{aligned} \quad (8)$$

$$= \left(\sum_{i < k} w_i\right)(\mu_n + \mu_g - 2\mu_\ell)w_k/2 \quad (9)$$

For items of larger rank than r_k , all of the ranks increase by w_k when we add the new item and μ_u increases by w_k , thus the sum of squares within the group is unchanged. The same derivation as above gives an update of

$$\begin{aligned} \sum_{i > k} w_i(r_i - \mu_n)^2 - \sum_{i > k} w_i(r_i - \mu_g)^2 &= \left(\sum_{i > k} w_i\right)[(\mu_u - \mu_n)^2 - ((\mu_u - w_k) - \mu_g)^2] \\ &= \left(\sum_{i > k} w_i\right)(\mu_n + z - 2\mu_u)(\mu_n - z) \end{aligned} \quad (10)$$

$$= \left(\sum_{i > k} w_i\right)(\mu_n + z - 2\mu_u)(-w_k/2) \quad (11)$$

$$z \equiv \mu_g + w_k$$

For items of tied rank, their rank increases by the same amount as the overall mean, and so their contribution to the total SS is unchanged. The final part of the update step is to add in the SS contributed by the new observation.

An observation is removed from the tree whenever the current time becomes less than the (start, stop] interval of the datum. The ranks for observations of lower risk are unchanged by the removal so equation (8) applies just as before, but with the new mean smaller than the old so the last term in equation (9) changes sign. For the observations of higher risk both the mean and the ranks change by w_k and equation (10) holds but with $z = \mu_0 - w_k$.

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 ascending time, with deaths preceding censorings.

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

wt case weight for the observation

sum scratch space, weights for each node of the tree: 3 values are for the node, all left children, and all right children

count the returned counts of concordant, discordant, tied on x, tied on time, and the variance

$\langle \text{concordance1} \rangle \equiv$

```
#include "survS.h"
SEXP concordance1(SEXP y, SEXP wt2, SEXP indx2, SEXP ntree2) {
    int i, j, k, index;
    int child, parent;
    int n, ntree;
    double *time, *status;
    double *twt, *nwt, *count;
    double vss, myrank, wsum1, wsum2, wsum3; /*sum of wts below, tied, above*/
    double lmean, umean, oldmean, newmean;

    double ndeath; /* weighted number of deaths at this point */

    SEXP count2;
    double *wt;
    int *indx;

    n = nrows(y);
    ntree = asInteger(ntree2);
    wt = REAL(wt2);
    indx = INTEGER(indx2);

    time = REAL(y);
    status = time + n;
    PROTECT(count2 = allocVector(REALSXP, 5));
    count = REAL(count2); /* count5 contains the information matrix */
    twt = (double *) R_alloc(2*ntree, sizeof(double));
    nwt = twt + ntree;
    for (i=0; i< 2*ntree; i++) twt[i] =0.0;
    for (i=0; i<5; i++) count[i]=0.0;
    vss=0;

     $\langle \text{concordance1-work} \rangle$ 

    UNPROTECT(1);
    return(count2);
}
```

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 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 $wt[i]*wt[j]$ to the appropriate count. Starting at the largest time (which is sorted last), walk through the tree.

- If it is a death time, we need to process all the deaths tied at this time.
 1. Add $wt[i] * wt[j]$ to the tied-on-time total, for all pairs i, j of tied times.
 2. The addition to tied-on-r will be the weight of this observation times the sum of weights for all others with the same risk score and a greater time, i.e., the weight found at `indx[i]` in the tree.
 3. Similarly for those with smaller or larger risk scores. First add in the children of this node. The left child will be smaller risk scores (and longer times) adding to the concordant pairs, the right child discordant. Then walk up the tree to the root. At each step up we add in data for the 'not me' branch. If we were the right branch (even number node) of a parent then when moving up we add in the left branch counts, and vice-versa.
- Now add this set of subject weights into the tree. The weight for a node is `nwt` and for the node and all its children is `twt`.

```

<concordance1-work>≡
for (i=n-1; i>=0; ) {
  ndeath =0;
  if (status[i]==1) { /* process all tied deaths at this point */
    for (j=i; j>=0 && status[j]==1 && time[j]==time[i]; j--) {
      ndeath += wt[j];
      index = indx[j];
      for (k=i; k>j; k--) count[3] += wt[j]*wt[k]; /* tied on time */
      count[2] += wt[j] * nwt[index]; /* tied on x */
      child = (2*index) +1; /* left child */
      if (child < ntree)
        count[0] += wt[j] * twt[child]; /*left children */
      child++;
      if (child < ntree)
        count[1] += wt[j] * twt[child]; /*right children */

      while (index >0) { /* walk up the tree */
        parent = (index-1)/2;
        if (index & 1) /* I am the left child */
          count[1] += wt[j] * (twt[parent] - twt[index]);
        else count[0] += wt[j] * (twt[parent] - twt[index]);
        index = parent;
      }
    }
  }
}

```

```

    }
else j = i-1;

/* Add the weights for these obs into the tree and update variance*/
for (; i>j; i--) {
    wsum1=0;
    oldmean = twt[0]/2;
    index = indx[i];
    nwt[index] += wt[i];
    twt[index] += wt[i];
    wsum2 = nwt[index];
    child = 2*index +1; /* left child */
    if (child < ntree) wsum1 += twt[child];

    while (index >0) {
        parent = (index-1)/2;
        twt[parent] += wt[i];
        if (!(index&1)) /* I am a right child */
            wsum1 += (twt[parent] - twt[index]);
        index=parent;
    }
    wsum3 = twt[0] - (wsum1 + wsum2); /* sum of weights above */
    lmean = wsum1/2;
    umean = wsum1 + wsum2 + wsum3/2; /* new upper mean */
    newmean = twt[0]/2;
    myrank = wsum1 + wsum2/2;
    vss += wsum1*(newmean+ oldmean - 2*lmean) * (newmean - oldmean);
    vss += wsum3*(newmean+ oldmean+ wt[i]- 2*umean) *(oldmean-newmean);
    vss += wt[i]* (myrank -newmean)*(myrank -newmean);
}
count[4] += ndeath * vss/twt[0];
}

```

The code for [start, stop) data is quite similar. 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.

```

<concordance1>+≡
SEXP concordance2(SEXP y,      SEXP wt2,  SEXP indx2, SEXP ntree2,
                  SEXP sortstop, SEXP sortstart) {
    int i, j, k, index;
    int child, parent;
    int n, ntree;
    int istart, iptr, jptr;
    double *time1, *time2, *status, dtime;
    double *twt, *nwt, *count;
    int *sort1, *sort2;

```

```

double vss, myrank;
double wsum1, wsum2, wsum3; /*sum of wts below, tied, above*/
double lmean, umean, oldmean, newmean;
double ndeath;
SEXP count2;
double *wt;
int *indx;

n = nrows(y);
ntree = asInteger(ntree2);
wt = REAL(wt2);
indx = INTEGER(indx2);
sort2 = INTEGER(sortstop);
sort1 = INTEGER(sortstart);

time1 = REAL(y);
time2 = time1 + n;
status= time2 + n;
PROTECT(count2 = allocVector(REALSXP, 5));
count = REAL(count2);
twt = (double *) R_alloc(2*ntree, sizeof(double));
nwt = twt + ntree;
for (i=0; i< 2*ntree; i++) twt[i] =0.0;
for (i=0; i<5; i++) count[i]=0.0;
vss =0;
<concordance2-work>

UNPROTECT(1);
return(count2);
}

```

The processing changes in 2 ways

- The loops go from 0 to $n - 1$ instead of $n - 1$ to 0. We need to use `sort1[i]` instead of `i` as the subscript for the `time2` and `wt` vectors. (The sort vectors go backwards in time.) This happens enough that we use a temporary variables `iptr` and `jptr` to avoid the double subscript.
- 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.

<concordance2-work>≡

```

istart = 0; /* where we are with start times */
for (i=0; i<n; ) {
    iptr = sort2[i]; /* In reverse death time order */
    ndeath =0;

```

```

if (status[iptr]==1) {
    /* Toss people out of the tree and update variance */
    dtime = time2[iptr];
    for (; istart < n && time1[sort1[istart]] >= dtime; istart++) {
        wsum1 =0;
        oldmean = twt[0]/2;
        jpتر = sort1[istart];
        index = indx[jptr];
        nwt[index] -= wt[jptr];
        twt[index] -= wt[jptr];
        wsum2 = nwt[index];
        child = 2*index +1; /* left child */
        if (child < ntree) wsum1 += twt[child];
        while (index >0) {
            parent = (index-1)/2;
            twt[parent] -= wt[jptr];
            if (!(index&1)) /* I am a right child */
                wsum1 += (twt[parent] - twt[index]);
            index=parent;
        }
        wsum3 = twt[0] - (wsum1 + wsum2);
        lmean = wsum1/2;
        umean = wsum1 + wsum2 + wsum3/2; /* new upper mean */
        newmean = twt[0]/2;
        myrank = wsum1 + wsum2/2;
        vss += wsum1*(newmean+ oldmean - 2*lmean) * (newmean-oldmean);
        oldmean -= wt[jptr]; /* the z in equations above */
        vss += wsum3*(newmean+ oldmean -2*umean) * (newmean-oldmean);
        vss -= wt[jptr]* (myrank -newmean)*(myrank -newmean);
    }

    /* Process deaths */
    for (j=i; j <n && status[sort2[j]]==1 && time2[sort2[j]]==dtime; j++) {
        jpتر = sort2[j];
        ndeath += wt[jptr];
        index = indx[jptr];
        for (k=i; k<j; k++) count[3] += wt[jptr]*wt[sort2[k]];
        count[2] += wt[jptr] * nwt[index]; /* tied on x */
        child = (2*index) +1; /* left child */
        if (child < ntree) count[0] += wt[jptr] * twt[child];
        child++;
        if (child < ntree) count[1] += wt[jptr] * twt[child];

        while (index >0) { /* walk up the tree */
            parent = (index-1)/2;
            if (index &1) /* I am the left child */

```

```

        count[1] += wt[jptr] * (tw[t[parent] - tw[t[index]]);
    else count[0] += wt[jptr] * (tw[t[parent] - tw[t[index]]);
    index = parent;
    }
    }
}
else j = i+1;

/* Add the weights for these obs into the tree and compute variance */
for (; i<j; i++) {
    wsum1 =0;
    oldmean = tw[0]/2;
    iptr = sort2[i];
    index = ind[x[iptr];
    nwt[index] += wt[iptr];
    tw[index] += wt[iptr];
    wsum2 = nwt[index];
    child = 2*index +1; /* left child */
    if (child < ntree) wsum1 += tw[child];
    while (index >0) {
        parent = (index-1)/2;
        tw[parent] += wt[iptr];
        if (!(index&1)) /* I am a right child */
            wsum1 += (tw[parent] - tw[index]);
        index=parent;
    }
    wsum3 = tw[0] - (wsum1 + wsum2);
    lmean = wsum1/2;
    umean = wsum1 + wsum2 + wsum3/2; /* new upper mean */
    newmean = tw[0]/2;
    myrank = wsum1 + wsum2/2;
    vss += wsum1*(newmean+ oldmean - 2*lmean) * (newmean-oldmean);
    vss += wsum3*(newmean+ oldmean +wt[iptr] - 2*umean) * (oldmean-newmean);
    vss += wt[iptr]* (myrank -newmean)*(myrank -newmean);
    }
count[4] += ndeath * vss/tw[0];
}

```

One last wrinkle is tied risk scores: they are all set to point to the same node of the tree. Here is the main routine.

```

<survConcordance>≡
survConcordance <- function(formula, data,
                             weights, subset, na.action) {
    Call <- match.call() # save a copy of of the call, as documentation

    m <- match.call(expand.dots=FALSE)

```

```

m[[1]] <- as.name("model.frame")
m$formula <- if(missing(data)) terms(formula, "strata")
                else                terms(formula, "strata", data=data)
m <- eval(m, sys.parent())
Terms <- attr(m, 'terms')

Y <- model.extract(m, "response")
if (!inherits(Y, "Surv")) {
  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")
}
n <- nrow(Y)

wt <- model.extract(m, 'weights')
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 <- m[[stemp$vars]]
  else strat <- strata(m[,stemp$vars], shortlabel=TRUE)
  Terms <- Terms[-stemp$terms]
}
else strat <- NULL

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

count <- survConcordance.fit(Y, x, strat, wt)
if (is.null(strat)) {
  concordance <- (count[1] + count[3])/sum(count[1:3])
  std.err <- count[5]/(2* sum(count[1:3]))
}
else {
  temp <- colSums(count)
  concordance <- (temp[1] + temp[3])/ sum(temp[1:3])
  std.err <- temp[5]/(2*sum(temp[1:3]))
}

fit <- list(concordance= concordance, stats=count, n=n,
           std.err=std.err, call=Call)
na.action <- attr(m, "na.action")
if (length(na.action)) fit$na.action <- na.action

oldClass(fit) <- 'survConcordance'
fit

```

```

}

print.survConcordance <- function(x, ...) {
  if(!is.null(c1 <- x$call)) {
    cat("Call:\n")
    dput(c1)
    cat("\n")
  }
  omit <- x$na.action
  if(length(omit))
    cat("  n=", x$n, " (", nprint(omit), ")\n", sep = "")
  else cat("  n=", x$n, "\n")
  cat("Concordance= ", format(x$concordance), " se= ", format(x$std.err),
      '\n', sep='')
  print(x$stats)

  invisible(x)
}

```

This part of the computation is a separate function, since it is also called by the coxph routines. Although we are very careful to create integers and/or doubles for the arguments to `.Call` I still wrap them in the appropriate `as.xxx` construction: “belt and suspenders”. Also, referring to the the mathematics many paragraphs ago, the C routine returns the variance of $(C - D)/2$ and we return the standard deviation of $(C - D)$. If this routine is called with all the x values identical, then C and D will both be zero, but the calculated variance of $C - D$ can be a nonzero tiny number due to round off error. Since this can cause a warning message from the `sqrt` function we check and correct this.

```

<survConcordance.fit>≡
  survConcordance.fit <- function(y, x, strata, weight) {
    <btree>

    docount <- function(stime, risk, wts) {
      if (attr(stime, 'type') == 'right') {
        ord <- order(stime[,1], -stime[,2])
        ux <- sort(unique(risk))
        n2 <- length(ux)
        index <- btree(n2)[match(risk[ord], ux)] - 1L
        .Call('concordance1', stime[ord,],
              as.double(wts[ord]),
              as.integer(index),
              as.integer(length(ux)))
      }
      else if (attr(stime, 'type') == "counting") {
        sort.stop <- order(-stime[,2], stime[,3])
        sort.start <- order(-stime[,1])
        ux <- sort(unique(risk))
      }
    }
  }

```

```

n2 <- length(ux)
index <- btree(n2)[match(risk, ux)] - 1L

.Call('concordance2', stime,
      as.double(wts),
      as.integer(index),
      as.integer(length(ux)),
      as.integer(sort.stop-1L),
      as.integer(sort.start-1L))
}
else stop("Invalid survival type for concordance")
}

if (missing(weight) || length(weight)==0)
  weight <- rep(1.0, length(x))
storage.mode(y) <- "double"

if (missing(strata) || length(strata)==0) {
  count <- docount(y, x, weight)
  if (count[1]==0 && count[2]==0) count[5]<-0
  else count[5] <- 2*sqrt(count[5])
  names(count) <- c("concordant", "discordant", "tied.risk", "tied.time",
                  "std(c-d)")
}
else {
  strata <- as.factor(strata)
  ustrat <- levels(strata)[table(strata) >0] #some strata may have 0 obs
  count <- matrix(0., nrow=length(ustrat), ncol=5)
  for (i in 1:length(ustrat)) {
    keep <- which(strata == ustrat[i])
    count[i,] <- docount(y[keep,,drop=F], x[keep], weight[keep])
  }

  count[,5] <- 2*sqrt(iffelse(count[,1]+count[,2]==0, 0, count[,5]))
  dimnames(count) <- list(ustrat, c("concordant", "discordant",
                                   "tied.risk", "tied.time",
                                   "std(c-d)"))
}
count
}

```

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 another ways, both of which are more sophisticated ways to deal with censoring. The individual curves are derived 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 subjects, 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 given

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.

cohort Should the program produce an overall survival curve (`cohort=T`) or separate estimates for each subject.

conditional chooses between the Hakulinen and conditional methods.

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.

npoints This argument is rarely used. If `times` is not given then output at `npoints` evenly spaced points spanning the range of `y`.

se.fit Add standard errors to the output.

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. In R the result has a class of `c('survexp', 'survfit')`; in Splus this needs to be a global setting using the `setOldClass` function.

```
<survexp>≡  
if (!is.R()) setOldClass(c('survexp', 'survfit')) #set up inheritance  
  
survexp <- function(formula, data,  
                    weights, subset, na.action, rmap,
```

```

    times, cohort=TRUE, conditional=FALSE,
    ratetable=survexp.us, scale=1, npoints, 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()
m <- match.call(expand.dots=FALSE)

# keep the first element (the call), and the following selected arguments
m <- m[c(1, match(c('formula', 'data', 'weights', 'subset', 'na.action'),
                  names(m), nomatch=0))]
m[[1]] <- as.name("model.frame")

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.

```

<surveyp-setup>+=
  rate <- attr(Terms, "specials")$ratetable
  if(length(rate) > 1)
    stop("Can have only 1 ratetable() call in a formula")
<surveyp-setup-rmap>

  if (is.R()) m <- eval(m, parent.frame())
  else      m <- eval(m, sys.parent())

<surveyp-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 not 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 <- attr(ratetable, "dimid")
else if(inherits(ratetable, "coxph")) {
  ## 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.

```

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

```

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(m)
if (n==0) stop("Data set has 0 rows")
weights <- model.extract(m, 'weights')
if (!is.null(weights)) 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.

```

<survexp-setup>+=
  Y <- model.extract(m, 'response')
  no.Y <- is.null(Y)
  if (!no.Y) {
    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)
    if (missing(times)) newtime <- sort(temp)
    else newtime <- sort(unique(c(times, temp[temp<max(times)])))
  }
  else conditional <- FALSE

```

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 matrix 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, m), stringsAsFactors=TRUE)
  if (is.ratetable(ratetable)) {
    israte <- TRUE
    if (no.Y) {
      if (missing(times))
        stop("There is no times argument, and no follow-up times are given in the formula")
      else newtime <- sort(unique(times))
      Y <- rep(max(times), n)
    }
    se.fit <- FALSE
    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(m[,rate]) != attr(ratetable$terms, 'term.labels')))
    stop("Unable to match new data to old formula")
  R <- model.matrix.coxph(ratetable, data=rdata)

  if (no.Y) {
    if (missing(se.fit)) se.fit <- TRUE
  }
  else se.fit <- FALSE
}
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 (!cohort) { #individual survival
  if (no.Y) stop("For non-cohort, an observation time must be given")
  if (israte)
    temp <- survexp.fit (1:n, R, Y, max(Y), TRUE, ratetable)
  else temp<- survexp.cfit(1:n, R, Y, FALSE, TRUE, ratetable, FALSE)
  xx <- temp$surv
  names(xx) <- row.names(m)
  na.action <- attr(m, "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 <- m[[ovars[i]]]

```

```

        ctemp <- class(temp)
        if (!is.null(ctemp) && ctemp=='tcut')
            stop("Can't use tcut variables in expected survival")
    }
    X <- strata(m[ovars])
}

#do the work
if (israte)
    temp <- survexp.fit(as.numeric(X), R, Y, newtime,
                       conditional, ratetable)
else {
    temp <- survexp.cfit(as.numeric(X), R, Y, conditional, FALSE,
                       ratetable, se.fit=se.fit)
    newtime <- temp$times
}

```

Now we need to package up the curves properly. Until such time as the program accepts Cox models with strata (on the some-day list), 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
    if (se.fit) err <- temp$se
}
else {
    if (israte) keep <- match(times, newtime)
    else {
        # The result is from a Cox model, and it's list of
        # times won't match the list requested in the user's call
        # Interpolate the step function, giving survival of 1 and
        # se of 0 for requested points that precede the Cox fit's
        # first downward step. The code is like summary.survfit.
        n <- length(newtime)
        keep <- approx(c(0, newtime), 0:n, xout=times,
                      method='constant', f=0, rule=2)$y
    }

    if (is.matrix(temp$surv)) {
        surv <- (rbind(1,temp$surv))[keep+1,,drop=FALSE]
        n.risk <- temp$n[pmax(1,keep),,drop=FALSE]
        if (se.fit) err <- (rbind(0,temp$se))[keep+1,,drop=FALSE]
    }
    else {

```

```

        surv <- (c(1,temp$surv))[keep+1]
        n.risk <- temp$n[pmax(1,keep)]
        if (se.fit) err <- (c(0,temp$se))[keep+1]
      }
      newtime <- times
    }
  newtime <- newtime/scale
  if (is.matrix(surv)) {
    dimnames(surv) <- list(NULL, levels(X))
    out <- list(call=call, surv=surv, n.risk=c(n.risk[,1]),
               time=newtime)
    if (se.fit) out$std.err <- err
  }
  else {
    out <- list(call=call, surv=c(surv), n.risk=c(n.risk),
               time=newtime)
    if (se.fit) out$std.err <- c(err)
  }
}

```

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$model <- m
  else {
    if (x) out$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'
  if (is.R()) class(out) <- c('survexp', 'survfit')
  else oldClass(out) <- c('survexp')
  out

```

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 explanation here.

```
<pyears-setup>≡
  expect <- match.arg(expect)
  call <- match.call()
  m <- match.call(expand.dots=FALSE)
  m <- m[c(1, match(c('formula', 'data', 'weights', 'subset', 'na.action'),
                  names(m), nomatch=0))]
  m[[1]] <- as.name("model.frame")

  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

if (is.R()) m <- eval(m, parent.frame())
else      m <- eval(m, sys.parent())

Y <- model.extract(m, '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")
  if (ncol(Y)==2 && any(Y[,2] <= Y[,1]))
    stop("stop time must be > start time")
}
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(m, '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 matrix 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, m), stringsAsFactors=TRUE)
  if (is.ratetable(ratetable)) {
    israte <- TRUE
    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("pyears cannot handle stratified Cox models")

    if (any(names(m[,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)
  for (i in 1:odim) {
    temp <- m[[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 February; 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 1/1/1960).

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)
}

```

```

cuts <- atts$cutpoints
if (is.null(atts$type)) {
  #old stlye table
  rfac <- atts$factor
  us.special <- (rfac >1)
}
else {
  rfac <- 1*(atts$type ==1)
  us.special <- (atts$type==4)
}
if (any(us.special)) { #special handling for US pop tables
  # Now, the 'entry' date on a US rate table is the number of days
  # since 1/1/1960, and the user data has been aligned to the
  # same system by match.ratetable and marked as "year".
  # The birth date is entry date - age in days (based on 1/1/1960).
  # I don't much care which date functions I use to do the arithmetic
  # below. Unfortunately R and Splus don't share one. My "date"
  # class is simple, but is also one of the earlier date class
  # attempts, has less features than others, and will one day fade
  # away; so I don't want to depend on it alone.
  #
  cols <- match(c("age", "year"), atts$dimid)
  if (any(is.na(cols)))
    stop("Ratetable does not have expected shape")
  if (exists("as.Date")) { # true for modern version of R
    bdate <- as.Date('1960/1/1') + (R[,cols[2]] - R[,cols[1]])
    byear <- format(bdate, "%Y")
    offset <- bdate - as.Date(paste(byear, "01/01", sep='/'),
                              origin="1960/01/01")
  }
  #else if (exists('month.day.year')) { # Splus, usually
  #   bdate <- R[,cols[2]] - R[,cols[1]]
  #   byear <- month.day.year(bdate)$year
  #   offset <- bdate - julian(1,1,byear)
  # }
  #else if (exists('date.mdy')) { # Therneau's date class is available
  #   bdate <- as.date(R[,cols[2]] - R[,cols[1]])
  #   byear <- date.mdy(bdate)$year
  #   offset <- bdate - mdy.date(1,1,byear)
  # }
  else stop("Can't find an appropriate date class\n")
  R[,cols[2]] <- R[,cols[2]] - offset

  # Doctor up "cutpoints" - only needed for 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)
  }
}

temp <- .C("pyears1",
          as.integer(n),
          as.integer(ncol(Y)),
          as.integer(is.Surv(Y)),
          as.double(Y),
          as.double(weights),
          as.integer(length(attrs$dim)),
          as.integer(rfac),
          as.integer(attrs$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(is.Surv(Y)) osize else 1),
          pexpect=double(osize),
          offtable=double(1),
          DUP=FALSE)[18:22]
}
else { #no expected
  temp <- .C('pyears2',
            as.integer(n),
            as.integer(ncol(Y)),
            as.integer(is.Surv(Y)),
            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(is.Surv(Y)) osize else 1),
        offtable=double(1)) [11:14]
    }

```

Create the output object.

```

<pyears-finish>≡
if (data.frame) {
  # Create a data frame as the output, rather than a set of
  # rate tables
  keep <- (temp$pyears >0) # what rows to keep in the output
  names(outdname) <- ovars
  if (length(outdname) ==1) {
    # if there is only one variable, the call to "do.call" loses
    # the variable name, since expand.grid returns a factor
    df <- data.frame((outdname[[1]])[keep],
                    pyears= temp$pyears[keep]/scale,
                    n = temp$pn[keep])
    names(df) <- c(names(outdname), 'pyears', 'n')
  }
  else {
    df <- cbind(do.call("expand.grid", outdname)[keep,],
                pyears= temp$pyears[keep]/scale,
                n = temp$pn[keep])
  }
  row.names(df) <- 1:nrow(df)
  if (has.ratetable) df$expected <- temp$pexpect[keep]
  if (expect=='pyears') df$expected <- df$expected/scale
  if (is.Surv(Y)) df$event <- temp$pcount[keep]

  out <- list(call=call,
              data= df, offtable=temp$offtable/scale)
  if (has.ratetable && !is.null(rtemp$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$offtable/scale)
  if (has.ratetable) {
    out$expected <- temp$pexpect
    if (expect=='pyears') out$expected <- out$expected/scale
    if (!is.null(rtemp$summ)) out$summary <- rtemp$summ
  }
  if (is.Surv(Y)) 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$offtable/scale)
  if (has.ratetable) {
    out$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 (is.Surv(Y))
    out$event <- array(temp$pcount, dim=odims, dimnames=outdname)
  }
na.action <- attr(m, "na.action")
if (length(na.action)) out$na.action <- na.action
if (model) out$model <- m
else {
  if (x) out$x <- X
  if (y) out$y <- Y
  }
oldClass(out) <- 'pyears'
out

```

7 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.

7.1 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.

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>≡
# $Id$
#
# 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 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')$strata
  coef <- object$coefficients
  intercept <- attr(Terms, "intercept")
  response <- attr(Terms, "response")
  weights <- object$weights
  if (is.null(weights)) weighted <- FALSE
```

```

  <rsr-dist>
  <rsr-data>
  <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$dist
  if (is.null(dd$itrans)) {
    itrans <- dtrans <-function(x)x
  }
  else {
    itrans <- dd$itrans
    dtrans <- dd$dtrans
  }
  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 <- model.frame(object)

  y <- object$y

```

```

if (is.null(y)) {
  y <- model.extract(mf, 'response')
  if (!is.null(dd$trans)) {
    tranfun <- dd$trans
    exactsurv <- y[,ncol(y)] ==1
    if (any(exactsurv)) logcorrect <-sum(log(dd$dtrans(y[exactsurv,1])))

    if (type=='interval') {
      if (any(y[,3]==3))
        y <- cbind(tranfun(y[,1:2]), y[,3])
      else y <- cbind(tranfun(y[,1]), y[,3])
    }
    else if (type=='left')
      y <- cbind(tranfun(y[,1]), 2-y[,2])
    else y <- cbind(tranfun(y[,1]), y[,2])
  }
  else {
    if (type=='left') y[,2] <- 2- y[,2]
    else if (type=='interval' && all(y[,3]<3)) y <- y[,c(1,3)]
  }
}

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$params)
    rr <- itrans(yhat0$center) - itrans(object$linear.predictor)
  }
  else {
    <rtr-deriv>
    <rtr-resid2>
  }

```

The matrix of derivatives is used in all of the other cases. The starting point is the `density` function of the distribution 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.

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

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]    #'
  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 calculate 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 *uncollapsed* 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
if (type=='dfbetas') rr <- rr %*% diag(1/sqrt(diag(vv)))
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
```

8 Competing risks

The competing risks routine is very general, allowing subjects to enter or exit states multiple times. For this reason I prefer the label “current prevalence” estimate, since it estimates what fraction of the subjects are in any given state across time. The easiest way to understand the estimate is to consider first the case of no censoring. In that setting the estimate of $F_k(t) = 1 - S_k(t)$ is the number of subjects in state k at time t 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 the as “cumulative incidence” curves, which is confusing, but the routine name `survfitCI` endures. The canonical call is

```
fit <- survfit(Surv(time, status, type='mstate') ~ sex, data=mine)
```

Optionally, there can be an `id` statement or `cluster` term to indicate a data set with multiple transitions per subject. A multi-state survival has a status variable with multiple levels, the first of which by default is censoring, and others indicating the type of transition that occurred. The result will be a matrix of survival curves, one for each event type. Subjects are assumed to start in a “null” state, which is not tabulated for survival. To change this behavior, give all subjects some other initial state.

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

```
<survfitCI>≡  
survfitCI <- function(X, Y, weights, id, istate,  
  type=c('kaplan-meier', 'fleming-harrington', 'fh2'),
```

```

        se.fit=TRUE,
        conf.int= .95,
        conf.type=c('log', 'log-log', 'plain', 'none'),
        conf.lower=c('usual', 'peto', 'modified')){

method <- match.arg(type)
#   error <- match.arg(error)
#   if (error != "inf")
#       warning("Only the infinitesimal jackknife error is supported for CI curves")
conf.type <- match.arg(conf.type)
conf.lower<- match.arg(conf.lower)

type <- attr(Y, "type")
if (type !='mright' && type!='mcounting' &&
    type != "right" && type != "counting")
    stop(paste("Cumulative incidence computation doesn't support \"", type,
              "\" survival data", sep=''))

n <- nrow(Y)
status <- Y[,ncol(Y)]
ncurve <- length(levels(X))

state.names <- attr(Y, "states")
if (missing(istate) || is.null(istate)) istate <- rep(0L, n)
else if (is.factor(istate) || is.character(istate)) {
    # Match levels with the survival variable
    temp <- as.factor(istate)
    state.names <- unique(c(attr(Y, "states"), levels(istate)))
    istate <- as.numeric(factor(as.character(istate), levels=state.names))
}
else if (!is.numeric(istate) || any(istate != floor(istate)))
    stop("istate should be a vector of integers or a factor")

if (length(id) ==0) id <- 1:n
# these next two lines should be impossible, since istate came from the data frame
if (length(istate) ==1) istate <- rep(istate,n)
if (length(istate) !=n) stop ("wrong length for istate")

states <- sort(unique(c(istate, 1:length(attr(Y, "states"))))) #list of all
<surfitCI-compute>

```

To make it easier to keep track of things in the computational kernel that does all the real work, we reset the states, initial state, and status vectors to all be integers 1, 2, ..., nstate, where “1” is the first state. The status vector will have values of 0 for censored. Per earlier discussion 1 will often be the unnamed initial state, which will later be dropped from the output. The statename vector is not modified.

```

<survfitCI>+≡
  if (any(states==0)) {
    state0 <- TRUE
    states <- states + 1
    istate <- istate + 1
    status <- ifelse(status==0, 0, status+1)
  }
  else state0 <- FALSE

  curves <- vector("list", ncurve)
  names(curves) <- levels(X)

  if (ncol(Y)==2) { # 1 transition per subject
    indx <- which(status == istate & status!=0)
    if (length(indx)) {
      warning("an observation transitions to it's starting state, ignored")
      status[indx] <- 0
    }
    if (length(id) && any(duplicated(id)))
      stop("Cannot have duplicate id values with (time, status) data")

    # dummy entry time that is < any event time
    entry <- rep(min(-1, 2*min(Y[,1])-1), n)
    for (i in levels(X)) {
      indx <- which(X==i)
      # temp <- docurve1(entry[indx], Y[indx,1], status[indx],
      #                   istate[indx], weights[indx], states,
      #                   id[indx])
      # curves[[i]] <- docurve2(entry[indx], Y[indx,1], status[indx],
      #                          istate[indx], weights[indx], states,
      #                          id[indx], se.fit)
    }
  }
  else {
    <survfitCI-startstop>
  }

  <survfitCI-finish>
}

```

In the multi-state case we can calculate the current prevalence vector $p(t)$ using the product-

limit form

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

Where p is a row vector and H is the multi-state hazard matrix. At each event time we define

$$H_{jk}(t) = \sum_i w_i dN_{ijk}(t) / \sum_i w_i Y_{ij}t$$

where N_{ijk} counts the number of observed transitions between state j and state k for subject i , $Y_{ij}(t)$ is 1 if subject i is in state j at time t , w_i is the weight for subject i , and $0/0$ is treated as 0. Row j of $H(t)$ describes the fate of those subjects in state j , going from time t to time $t + 0$. The diagonal elements of H are set so that 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.

A complementary formula is the hazard based calculation

$$S(t) = I - P = \exp^{-A(t)}$$

where \exp is the matrix exponential and A is the cumulative hazard matrix. The matrix $A(t)$ has off diagonal elements $\sum_{s \leq t} H(s)$, but the diagonal is chosen to give row sums of 0. This collapses to $\exp(-\Lambda(t))$ in the alive/dead case. The analog of the fh2 estimate comes from treating tied event times of the same type (same states) as sequential. The variance for these estimates is harder, so we haven't pursued it in the code yet.

A robust variance for the product-limit estimate is based on the chain rule. Consider the n by k matrix of per subject influence values

$$\begin{aligned} U_{ik}(t) &= \frac{\partial p_k(t)}{\partial w_i} \\ &= \frac{\partial [p(t-)H_{.k}(t)]}{\partial w_i} \end{aligned} \tag{12}$$

$$= U_{i.(t-)}H_{.k}(t) + p(t-)\frac{\partial H_{.k}(t)}{\partial w_i} \tag{13}$$

$$\frac{\partial H_{jk}(t)}{\partial w_i} = \begin{cases} (dN_{ijk}(t) - Y_{ij}(t)H_{jk})/n_j(t) & j \neq k \\ (-dN_{ij.}(t) - Y_{ij}(t)(H_{jj} - 1))/n_j(t) & j = k \end{cases} \tag{14}$$

where $H_{.k}$ is the k th column of H and $n_j(t) = \sum_i Y_{ij}(t)w_i$ is the weighted number of subjects in state j . Equation (12) replaces $p(t)$ with the last step of the computation that created it. The next writes this out carefully using the chain rule, leading to an recursive equation. The first term of (13) is the formula for ordinary matrix multiplication. In equation (14) the derivative of H with respect to subject i will be a matrix which is non-zero only for the row corresponding to the current state of the subject. (I've skipped some intermediate steps in the derivation of (14), "left as an exercise for the reader").

The weighted sum of each column of U must 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 $U'WU$ where W is a diagonal matrix of case weights, but we currently don't report that back.

Below is the function for a single curve. For the status variable a value of 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 subgroup might not have them all) which lead to 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. 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 at some given time and `dindx` for the corresponding subjects.

```

(survfitCI-compute-old)≡
docurve1 <- function(entry, etime, status, ystate, wt, states, id) {
  #
  # round off error can cause trouble: if two times are within machine
  # precision then "unique(time)" and the "table" command may differ
  # solve this by using creating a factor
  ftime <- factor(etime)
  ntime <- length(levels(ftime))
  timeset <- as.numeric(levels(ftime))
  ftime <- as.numeric(ftime)

  nstate <- length(states)
  Pmat <- matrix(0., nrow= ntime, ncol=nstate)
  vP <- Pmat #variance
  A <- array(0., dim=c(nstate, nstate, ntime))
  uid <- sort(unique(id))
  U <- matrix(0., length(uid), nstate) #one row per subject
  P <- as.vector(tapply(wt, factor(ystate, levels=states), sum) / sum(wt))
  P <- Pmat[1,] <- ifelse(is.na(P), 0, P)
  cstate <- ystate[match(uid, id)] #current state for each observation

  nrisk <- integer(ntime) #to be returned
  wrisk <- double(ntime) #weighted number at risk
  nevent <- table(ftime, status>0)
  for (i in 1:ntime) {
    atrisk <- (ftime >=i & timeset[i] > entry)
    nrisk[i] <- sum(atrisk)
    wrisk[i] <- sum(wt[atrisk])
    tiedtime <- (ftime==i)
    if (nevent[i,2] ==0) { # all censored here
      Pmat[i,] <- P
      if (i>1) {
        A[,,i] <- A[,,i-1]
        vP[i,] <- vP[i-1,]
      }
    }
  }
}

```

```

else {
  # do real work
  # A bit of nuisance is to force tapply to give totals for all states
  aindx <- match(id[atrisk], uid) #the id pointer for those at risk
  ns <- as.vector(tapply(wt[atrisk], factor(cstate[aindx], levels=states),sum))
  dead <- which(tiedtime & status >0) #the events at this time
  dindx <- match(id[dead], uid)
  nevent[i] <- length(dead)

  H <- tapply(wt[dead], list(factor(cstate[dindx], levels=states),
                             factor(status[dead], levels=states)),sum)/ns
  H <- ifelse(is.na(H), 0, H) # H has NA for combinations with no representatives
  diag(H) <- 1- rowSums(H)

  H2 <- H
  diag(H2) <- diag(H2) -1 #version of H needed for U and A, rows sum to 0
  if (i==1) A[,1] <- H2
  else A[,i] <- A[,i-1] + H2
  newstate <- status[dead] # where the transitions go, will never be 0
  oldstate <- cstate[dindx] # where they came from
  U <- U%*%H #first part of update

  U[aindx,] <- U[aindx,] - (P*H2/ns)[cstate[aindx], ]
  temp <- P[oldstate]/ns[oldstate] #the extra update for the events
  U[cbind(dindx, oldstate)] <- U[cbind(dindx, oldstate)] - temp
  U[cbind(dindx, newstate)] <- U[cbind(dindx, newstate)] + temp
  cstate[dindx] <- newstate
  P <- Pmat[i,] <- c(P %*% H)
  vP[i,] <- colSums(wt[match(uid, id)]*U*U)
}
}
list(time =as.vector(timeset), pmat=Pmat, std=sqrt(vP),
      n.event= as.vector(nevent[,2]), n.risk= as.vector(nrisk),
      w.risk=wrisk, cumhaz=A)
}

```

The above function was used to work through all of my test cases, but is too slow in large data sets. Rewrite it using underlying C-code, but retain the former one for debugging purposes. It appears at the end of this chapter.

The setup for (start, stop] data is a bit more work. We want to ensure that a subject's weight is fixed, that they have a continuous period of observation, and that they don't transfer from a state to itself.

```

<survfitCI-startstop>≡
  if (missing(id) || is.null(id))
    stop("the id argument is required for start:stop data")

```

```

indx <- order(id, Y[,2]) #ordered event times within subject
indx1 <- c(NA, indx) #a pair of lagged indices
indx2 <- c(indx, NA)
same <- (id[indx1] == id[indx2] & !is.na(indx1) & !is.na(indx2)) #indx1, indx2= same id?
if (any(same & X[indx1] != X[indx2])) {
  who <- 1 + min(which(same & X[indx1] != X[indx2]))
  stop("subject is in two different groups, id ", (id[indx1])[who])
}
if (any(same & Y[indx1,2] != Y[indx2,1])) {
  who <- 1 + min(which(same & Y[indx1,2] != Y[indx2,1]))
  stop("gap in follow-up, id ", (id[indx1])[who])
}
if (any(Y[,1] == Y[,2]))
  stop("cannot have start time == stop time")

if (any(same & Y[indx1,3] == Y[indx2,3] & Y[indx1,3] != 0)) {
  who <- 1 + min(which(same & Y[indx1,1] != Y[indx2,2]))
  stop("subject changes to the same state, id ", (id[indx1])[who])
}
if (any(same & weights[indx1] != weights[indx2])) {
  who <- 1 + min(which(same & weights[indx1] != weights[indx2]))
  stop("subject changes case weights, id ", (id[indx1])[who])
}

# We only want to pay attention to the istate variable for the very first
# observation of any given subject, but the program logic does better with
# a full one. So construct one that will do this
indx <- order(Y[,2])
uid <- unique(id)
temp <- (istate[indx])[match(uid, id[indx])] #first istate for each subject
istate <- temp[match(id, uid)] #replicate it to full length

# 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)
}

<survfitCI-finish>≡
# Turn the result into a survfit type object
grabit <- function(clist, element) {
  temp <- (clist[[1]][[element]])
  if (is.matrix(temp)) {

```

```

        nc <- ncol(temp)
        matrix(unlist(lapply(clist, function(x) t(x[[element]]))),
              byrow=T, ncol=nc)
      }
    else as.vector(unlist(lapply(clist, function(x) x[element])))
  }
kfit <- list(n =      as.vector(table(X)),
           time =    grabit(curves, "time"),
           n.risk=   grabit(curves, "n.risk"),
           n.event=  grabit(curves, "n.event"),
           n.censor=grabit(curves, "n.censor"),
           prev      = grabit(curves, "pmat"))
nstate <- length(states)
kfit$cumhaz <- array(unlist(lapply(curves, function(x) x$cumhaz)),
                   dim=c(nstate, nstate, length(kfit$time)))
if (length(curves) >1)
  kfit$strata <- unlist(lapply(curves, function(x) length(x$time)))
if (se.fit) kfit$std.err <- grabit(curves, "std")

# if state 0 was present, remove it
if (state0) {
  kfit$prev <- kfit$prev[,-1]
  if (se.fit) kfit$std.err <- kfit$std.err[,-1]
}

```

Add the confidence bands. The idea is modeled on `survfitKM` but with the important differences that we are dealing with P instead of S , and the “modified lower limit” logic does not apply. We make the assumption that $\log(1 - P)$ will have better CI behavior than P , with standard error of $rmse(P)/(1 - P)$.

```

<survfitCI-finish>+=
#
# Last bit: add in the confidence bands:
# modeled on survfit.km, though for P instead of S
#
#
if (se.fit) {
  std.err <- kfit$std.err
  zval <- qnorm(1- (1-conf.int)/2, 0,1)
  surv <- 1-kfit$prev

  if (conf.type=='plain') {
    temp <- zval* std.err
    kfit <- c(kfit, list(lower =pmax(kfit$prev-temp, 0),
                          upper=pmin(kfit$prev+temp, 1),
                          conf.type='plain', conf.int=conf.int))
  }
}

```

```

if (conf.type=='log') {
  #avoid some "log(0)" messages
  xx <- ifelse(kfit$prev==1, 1, 1- kfit$prev)

  temp1 <- ifelse(surv==0, NA, exp(log(xx) + zval* std.err/xx))
  temp2 <- ifelse(surv==0, NA, exp(log(xx) - zval* std.err/xx))
  kfit <- c(kfit, list(lower=pmax(1-temp1,0), upper= 1- temp2,
                        conf.type='log', conf.int=conf.int))
}

if (conf.type=='log-log') {
  who <- (surv==0 | surv==1) #special cases
  temp3 <- ifelse(surv==0, NA, 1)
  xx <- ifelse(who, .1,kfit$surv) #avoid some "log(0)" messages
  temp1 <- exp(-exp(log(-log(xx)) + zval*std.err/(xx*log(xx))))
  temp1 <- ifelse(who, temp3, temp1)
  temp2 <- exp(-exp(log(-log(xx)) - zval*std.err/(xx*log(xx))))
  temp2 <- ifelse(who, temp3, temp2)
  kfit <- c(kfit, list(lower=1-temp1, upper=1-temp2,
                        conf.type='log-log', conf.int=conf.int))
}
}

```

```

kfit$states <- state.names
kfit$type <- attr(Y, "type")
kfit

```

The updated docurve function is here

<survfitCI-compute>≡

```

docurve2 <- function(entry, etime, status, istate, wt, states, id, se.fit) {
  #
  # round off error can cause trouble, if two times are within machine
  # precision
  # solve this by using creating a factor
  ftime <- factor(c(entry,etime))
  ltime <- levels(ftime)
  ftime <- matrix(as.integer(ftime), ncol=2)
  n <- length(entry)
  timeset <- as.numeric(ltime[sort(unique(ftime[,2]))]) #unique event times

  nstate <- length(states)
  uid <- sort(unique(id))
  P <- as.vector(tapply(wt, factor(istate, levels=states), sum) / sum(wt))
  P <- ifelse(is.na(P), 0, P) # initial probability distribution
  cstate <- istate[match(uid, id)] #current state for each observation
}

```

```

storage.mode(wt) <- "double" # just in case someone had integer weights
storage.mode(cstate) <- "integer"
storage.mode(status) <- "integer"
# C code has 0 based subscripts
fit <- .Call("survfitci", ftime,
             order(ftime[,1]) - 1L,
             order(ftime[,2]) - 1L,
             length(timeset),
             status,
             cstate - 1L,
             wt,
             match(id, uid) -1L,
             P, as.integer(se.fit))
if (se.fit)
  list(time=timeset, pmat=t(fit$p), std=sqrt(t(fit$var)),
        n.risk = colSums(fit$nrisk),n.event = fit$nevent,
        n.censor=fit$ncensor,
        cumhaz=array(fit$cumhaz, dim=c(nstate,nstate, length(timeset))))
else list(time=timeset, pmat=t(fit$p),
          n.risk = colSums(fit$nrisk),n.event = fit$nevent,
          n.censor=fit$ncensor,
          cumhaz=array(fit$cumhaz, dim=c(nstate,nstate, length(timeset))))
}

```

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

First define a function that allocates a ragged array. The code has many matrices, and it is easier to use `x[i][j]` notation for them.

```

<survfitci>≡
#include "survS.h"

<survfitci-dmatrix>
SEXP survfitci(SEXP ftime2, SEXP sort12, SEXP sort22, SEXP ntime2,
              SEXP status2, SEXP cstate2, SEXP wt2, SEXP id2,
              SEXP p2, 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.

sefit If 1 then do the se computation, otherwise forget it.

The local `dmatrix2` function makes it easier to declare ragged arrays, which allows for the nice `x[i][j]` notation for arrays.

```
<survfitci-dmatrix>≡
/* allocate a ragged array of a given number of rows and columns */
static double **dmatrix2(int nrow, int ncol) {
    int i;
    double **mat;
    double *d;

    mat = (double **) R_alloc(nrow, sizeof(double *));
    d = (double *) R_alloc(nrow*ncol, sizeof(double));
    for (i=0; i<nrow; i++) {
        mat[i] = d;
        d += ncol;
    }
    return(mat);
}
```

Declare all of the variables.

```
<survfitci-declare>≡
int i, j, k, kk; /* generic loop indices */
int ck, itime, eptr; /*specific indices */
int ctime; /*current time of interest, in the main loop */
int nprotect; /* number of protect calls issued */
int oldstate, newstate; /*when changing state */

double temp, *temp2; /* scratch */
double *p; /* current prevalence vector */
double **hmat; /* hazard matrix at this time point */
double **umat; /* per subject leverage at this time point */
```

```

int *atrisk;          /* 1 if the subject is currently at risk */
int  *ns;            /* number curently in each 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 */
int *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 */
double *wt;         /* weight for each observation */
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", "prev",
                       "cumhaz", "var", ""};
SEXP pmat2, vmat2, cumhaz2; /*list components */
SEXP nevent2, ncensor2, nrisk2;
double *pmat, *vmat, *cumhaz;
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);
n  = LENGTH(sort12);
PROTECT(cstate2 = duplicate(cstate2));
cstate = INTEGER(cstate2);
entry= INTEGER(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 */

```

```

sefit = asInteger(sefit2);

/* allocate space for the output objects */
PROTECT(pmat2 = allocMatrix(REALSXP, nstate, ntime));
pmat = REAL(pmat2);
if (sefit >0)
    PROTECT(vmat2 = allocMatrix(REALSXP, nstate, ntime));
else PROTECT(vmat2 = allocMatrix(REALSXP, 1, 1)); /* dummy object */
vmat = REAL(vmat2);
PROTECT(nevent2 = allocVector(INTSXP, ntime));
nevent = INTEGER(nevent2);
PROTECT(ncensor2= allocVector(INTSXP, ntime));
ncensor = INTEGER(ncensor2);
PROTECT(nrisk2 = allocMatrix(INTSXP, nstate, ntime));
nrisk = INTEGER(nrisk2);
PROTECT(cumhaz2= allocVector(REALSXP, nstate*nstate*ntime));
cumhaz = REAL(cumhaz2);
nprotect = 8;

/* allocate space for scratch vectors */
ws = (double *) R_alloc(2*nstate, sizeof(double));
temp2 = ws + nstate;
ns = (int *) R_alloc(nstate, sizeof(int));
atrisk = (int *) R_alloc(nperson, sizeof(int));
wtp = (double *) R_alloc(nperson, sizeof(double));
hmat = (double**) dmatrix2(nstate, nstate);
if (sefit >0) umat = (double**) dmatrix2(nperson, nstate);
chaz = (double**) dmatrix2(nstate, nstate);

/* R_alloc does not zero allocated memory */
for (i=0; i<nstate; i++) {
    ws[i] =0;
    ns[i] =0;
    for (j=0; j<nstate; j++) {
        hmat[i][j] =0;
        chaz[i][j] =0;
    }
    if (sefit) {for (j=0; j<nperson; j++) umat[j][i]=0;}
}
for (i=0; i<nperson; i++) atrisk[i] =0;

```

The primary loop of the program walks along the `sort2` vector, with one pass through the loop for each unique event time. Observations are at risk in the interval (entry, event], note the round and square brackets, so we need `entry < ctime <= event`, 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. then remove old one. 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, which index by unique event time `itime`
- The `n` observations (variables `entry`, `event`, `sort1`, `sort2`, `status`, `wt`, `id`)
- The `nperson` individual subjects (variables `cstate`, `atrisk`)
- The `nstate` states (variables `hmat`, `p`)

```

<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 the jj element is set to 1. A second version which we call H2 has 1 subtracted from each diagonal and so that the 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 */
nevent[itime] =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]];
      nevent[itime]++;
      wevent += wt[k];
      hmat[oldstate][newstate] += wt[k];
    }
    else ncensor[itime]++;
  }
  else break;
}

if (nevent[itime]> 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 , which has `nperson` rows and `nstate` columns. It has 3 steps. Refer to equation (14) for the mathematical details.

1. The entire matrix is multiplied by H .
2. Consider the scaled matrix J whose k th row is the matrix $H2$ scaled by the value $p[k]/ws[k]$. (Probability of being in the state divided by the weighted number in the state). If subject i is currently at risk and currently in state k , then row k of J is subtracted from $U[i,]$.
3. For each subject i who had an event at this time and went from state j to state k , $U[i,j]$ will decrease by $p[j]/ws[j]$ and $U[i,k]$ will increase by the same amount.

If standard errors are not needed we can skip this calculation, which speeds up the code considerably.

```

<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];
}

/* Update U, part 2, subtract from everyone at risk
   For this I need H2 */
for (j=0; j<nstate; j++) hmat[j][j] -= 1;
for (j=0; j<nperson; j++) {
    if (atrisk[j]==1) {
        kk = cstate[j];
        for (k=0; k<nstate; k++)
            umat[j][k] -= (p[kk]/ws[kk])* hmat[kk][k];
    }
}

/* Update U, part 3. An addition for each event */
for (j=i; j<n; j++) {

```

```

k = sort2[j];
if (etime[k] == ctime) {
  if (status[k] >0) {
    kk = id[k]; /* row number in U */
    oldstate= cstate[kk];
    newstate= status[k] -1;
    umat[kk][oldstate] -= p[oldstate]/ws[oldstate];
    umat[kk][newstate] += p[oldstate]/ws[oldstate];
  }
}
else break;
}

```

Now update the cumulative hazard by adding H2 to it, and update p to pH . If sefit is 1 then H has already been transformed to H2 form.

```

<survfitci-compute-P>≡
/* Finally, update chaz and p. */
for (j=0; j<nstate; j++) {
  if (sefit ==0) hmat[j][j] -= 1; /* conversion to H2*/
  for (k=0; k<nstate; k++) chaz[j][k] += hmat[j][k];

  hmat[j][j] +=1; /* change from H2 to H */
  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++ = p[j];
  *nrisk++ = ns[j];
  for (k=0; k<nstate; k++) *cumhaz++ = chaz[k][j];
  temp=0;
  if (sefit >0) {
    for (k=0; k<nperson; k++)
      temp += wtp[k] * umat[k][j]*umat[k][j];
    *vmat++ = temp;
  }
}

```

```

<survfitci-return>≡
/* return a list */
PROTECT(rlist=mkNamed(VECSXP, rnames));
SET_VECTOR_ELT(rlist, 0, nrisk2);
SET_VECTOR_ELT(rlist, 1, nevent2);

```

```

SET_VECTOR_ELT(rlist, 2, ncensor2);
SET_VECTOR_ELT(rlist, 3, pmat2);
SET_VECTOR_ELT(rlist, 4, cumhaz2);
SET_VECTOR_ELT(rlist, 5, vmat2);
UNPROTECT(nprotect +1);
return(rlist);

```

8.1 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. If there is only one subscript it is preferentially associated with the strata, if there is no strata argument `i` will associate with the columns. If there are two subscripts the first goes with the strata. The little `nmatch` function allow the user to use either names or integer indices. The drop argument is important when strata get subscripted such that only one row remains and there are multiple columns: in that case we do *not* want to lose the matrix nature of the result as it will lead to an invalid object. Otherwise we can drop columns freely

(survfitms-subscript)≡

```

" [.survfitms" <- function(x, ..., drop=TRUE) {
  nmatch <- function(indx, target) {
    temp <- 1:length(target)
    names(temp) <- target
    temp[indx]
  }

  if (missing(..1)) i<- NULL else i <- sort(..1)
  if (missing(..2)) j<- NULL else j <- ..2
  if (is.null(x$strata)) {
    if (is.matrix(x$prev)) {
      # No strata, but a matrix of prevalence values
      indx <- nmatch(i, x$states)
      if (any(is.na(indx)))
        stop("unmatched subscript", i[is.na(indx)])
      else i <- as.vector(indx)
      x$states <- x$states[i]

      if (nrow(x$prev)==1 && length(i) > 1) drop<- FALSE
      x$prev <- x$prev[,i,drop=drop]
      x$cumhaz <- x$cumhaz[i,i,, drop=drop]
    }
  }
}

```

```

        if (!is.null(x$std.err)) x$std.err <- x$std.err[,i,drop=drop]
        if (!is.null(x$upper)) x$upper <- x$upper[,i,drop=drop]
        if (!is.null(x$lower)) x$lower <- x$lower[,i,drop=drop]
    }
    else warning("Survfit object has only a single survival curve")
  }
else {
  if (is.null(i)) keep <- seq(along.with=x$time) # rows to keep
  else {
    indx <- nmatch(i, names(x$strata)) #strata to keep
    if (any(is.na(indx)))
      stop(paste("strata",
                 paste(i[is.na(indx)], collapse=' '),
                 'not matched'))
    strat <- rep(names(x$strata), x$strata)
    keep <- seq(along.with=strat)[strat %in% names(indx)]
    if (length(indx) <=1) x$strata <- NULL
    else x$strata <- x$strata[indx]

    x$n      <- x$n[indx]
    x$time   <- x$time[keep]
    x$n.risk <- x$n.risk[keep]
    x$n.event <- x$n.event[keep]
    x$n.censor <- x$n.censor[keep]
  }
  if (is.matrix(x$prev)) {
    if (length(keep) <2) drop <- FALSE
    if (is.null(j)) { #only subscript rows (strata)
      x$prev <- x$prev[keep,,drop=drop]
      x$cumhaz <- x$cumhaz[,keep, drop=drop]
      if (!is.null(x$std.err))
        x$std.err <- x$std.err[keep,,drop=drop]
      if (!is.null(x$upper)) x$upper <- x$upper[keep,,drop=drop]
      if (!is.null(x$lower)) x$lower <- x$lower[keep,,drop=drop]
    }
    else { #subscript both rows (strata) and columns (states)
      indx <- nmatch(j, x$states)
      if (any(is.na(indx)))
        stop("unmatched subscript", j[indx])
      else j <- as.vector(indx)
      x$states <- x$states[j]
      x$prev <- x$prev[keep,j, drop=drop]
      x$cumhaz <- x$cumhaz[j,j,keep, drop=drop]
      if (!is.null(x$std.err)) x$std.err <- x$std.err[keep,j,drop=drop]
      if (!is.null(x$upper)) x$upper <- x$upper[keep,j, drop=drop]
      if (!is.null(x$lower)) x$lower <- x$lower[keep,j, drop=drop]
    }
  }
}

```

```

    }
  }
else {
  x$prev <- x$prev[keep]
  x$cumhaz <- x$cumhaz[keep]
  if (!is.null(x$std.err)) x$std.err <- x$std.err[keep]
  if (!is.null(x$upper)) x$upper <- x$upper[keep]
  if (!is.null(x$lower)) x$lower <- x$lower[keep]
}
}
x
}

```

The `summary.survfit` and `summary.survfitms` functions share a large amount of code. Both are included here in order to have common source for the most subtle block of it, which has to do with selecting intermediate time points.

```

<survfitms-summary>≡
summary.survfit <- function(object, times, censored=FALSE,
                             scale=1, extend=FALSE,
                             rmean=getOption('survfit.rmean'),
                             ...) {
  fit <- object
  if (!inherits(fit, 'survfit'))
    stop("summary.survfit can only be used for survfit objects")

  # The print.rmean option is depreciated, still paid
  # attention to in print.survfit, but ignored here
  if (is.null(rmean)) rmean <- "none"

  temp <- survmean(fit, scale=scale, rmean)
  table <- temp$matrix #for inclusion in the output list
  rmean.endtime <- temp$end.time

  # The fit$surv object is sometimes a vector and sometimes a
  # matrix. We calculate row indices first, and then deal
  # with the cases at the end.
  nsurv <- if (is.matrix(fit$surv)) nrow(fit$surv) else length(fit$surv)
  if (is.null(fit$strata)) {
    nstrat <- 1
    stemp <- rep(1L, nsurv)
    strata.names <- ""
  }
  else {
    nstrat <- length(fit$strata)
    stemp <- rep(1:nstrat, fit$strata)
  }
}

```

```

    strata.names <- names(fit$strata)
  }

  <survsum-findrows>

  # Create an output structure
  if (length(indx1)==length(fit$time) && all(indx1 == seq(along=fit$time))) {
    temp <- object #no change
    temp$time <- temp$time/scale
    temp$table <- table
    if (!is.null(temp$strata))
      temp$strata <- factor(stemp, labels=strata.names)
  }
  else if (missing(times)) { #default censor=FALSE case
    temp <- object
    temp$time <- temp$time[indx1]/scale
    temp$table <- table
    for (j in c("n.risk", "n.event", "n.censor", "n.enter",
               "surv", "std.err", "lower", "upper")) {
      zed <- temp[[j]]
      if (!is.null(zed)) {
        if (is.matrix(zed)) temp[[j]] <- zed[indx1,,drop=FALSE]
        else temp[[j]] <- zed[indx1]
      }
    }
    if (!is.null(temp$strata))
      temp$strata <- factor(stemp[indx1], levels=1:nstrat,
                          labels=strata.names)
  }
  else { #times argument was given
    temp <- list(n=object$n, time=times/scale,
               n.risk=n.risk, n.event=n.event,
               conf.int=fit$conf.int, type=fit$type, table=table)
    if (!is.null(n.censor)) temp$n.censor <- n.censor
    if (!is.null(n.enter)) temp$n.enter <- n.enter
    if (!is.null(fit$start.time)) temp$start.time <- fit$start.time

    # why the rbind? The user may have specified a time point before
    # the first event, and indx1=1 indicates that case
    if (is.matrix(fit$surv)) {
      temp$surv <- rbind(1, fit$surv)[indx1,,drop=FALSE]
      if (!is.null(fit$std.err))
        temp$std.err <- rbind(0, fit$std.err)[indx1,,drop=FALSE]
      if (!is.null(fit$lower)) {
        temp$lower <- rbind(1, fit$lower)[indx1,,drop=FALSE]
      }
    }
  }
}

```

```

        temp$upper <- rbind(1, fit$upper)[indx1,,drop=FALSE]
    }
}
else {
    temp$surv <- c(1, fit$surv)[indx1]
    if (!is.null(fit$std.err)) temp$std.err <- c(0,fit$std.err)[indx1]
    if (!is.null(fit$lower)) {
        temp$lower <- c(1, fit$lower)[indx1]
        temp$upper <- c(1, fit$upper)[indx1]
    }
}
}
if (!is.null(fit$strata)) {
    scount <- unlist(lapply(newtimes, length))
    temp$strata <- factor(rep(1:nstrat, scount), levels=1:nstrat,
                        labels=strata.names)
}

if (length(rmean.endtime)>0 && !is.na(rmean.endtime))
    temp$rmean.endtime <- rmean.endtime

temp$call <- fit$call
if (!is.null(fit$na.action)) temp$na.action <- fit$na.action

}
if (!is.null(temp$std.err))
    temp$std.err <- temp$std.err*temp$surv #std error of the survival curve
class(temp) <- 'summary.survfit'
temp
}

Grab rows: if there is no times argument it is easy
<survsum-findrows>≡
if (missing(times)) {
    # just pick off the appropriate rows of the output
    if (censored) indx1 <- seq(along=fit$time)
    else indx1 <- which(fit$n.event>0)
}

```

This second case is actual work, since may involve “in between” points in the curves. Let’s say that we have a line in the data for times 1,2, 5, and 6, and 8 and a user chose `times=c(3,5, 9)`. At time 3 we have

- `nrisk[3]` = value at the next time point $i=3$
- `nevent[1] + nevent[2]` = value since last printout line. However, if there are multiple strata the curves for all strata are laid end to end in a single vector; our first row for a curve needs to use all events since the start of the curve.

- ncensor works like nevent
- survival[2] = survival at the last time point $j=3$

At time 5 we pick values directly off the data, since we match. At time 9 we report nothing if `extend` is `FALSE`, or the value at the end of the curve. In this case we need to calculate the number at risk ourselves, however. This logic works out best if we do it curve by curve.

```

<survsum-findrows>+=
else {
  # The one line function below might be opaque (even to me) --
  # For n.event, we want to know the number since the last chosen
  # printout time point. Start with the curve of cumulative
  # events at c(0, stime) (the input time points), which is
  # the cumsum below; pluck off the values corresponding to our
  # time points, the [x] below; then get the difference since the
  # last chosen time point (or from 0, for the first chosen point).
  cfun <- function(x, data) diff(c(0, cumsum(c(0,data)))[x])

  # Process the curves one at a time,
  # adding the results for that curve onto a list, so the
  # number of events will be n.enter[[1]], n.enter[[2]], etc.
  # For the survival, stderr, and confidence limits it suffices
  # to create a single list 'indx1' containing a subscripting vector
  indx1 <- n.risk <- n.event <- newtimes <- vector('list', nstrat)
  n.enter <- vector('list', nstrat)
  n.censor<- vector('list', nstrat)
  n <- length(stemp)
  for (i in 1:nstrat) {
    who <- (1:n)[stemp==i] # the rows of the object for this strata
    stime <- fit$time[who]

    # First, toss any printing times that are outside our range
    if (is.null(fit$start.time)) mintime <- min(stime, 0)
    else mintime <- fit$start.time
    ptimes <- times[times >= mintime]

    if (!extend) {
      maxtime <- max(stime)
      ptimes <- ptimes[ptimes <= maxtime]
    }

    newtimes[[i]] <- ptimes

    # If we tack a -1 onto the front of the vector of survival
    # times, then indx1 is the subscript for that vector

```

```

# corresponding to the list of "ptimes". If the input
# data had stime=c(10,20) and ptimes was c(5,10,15,20),
# the result would be 1,2,2,3.
# For n.risk we want a slightly different index: 2,2,3,3.
# "In between" times point to the next higher index for n.risk,
# but the next lower one for survival. (Survival drops at time t,
# the n.risk immediately afterwards at time t+0: you were at
# risk just before you die, but not a moment after). The
# extra point needs to be added at the end.
#
n.time <- length(stime) #number of points
temp1 <- approx(c(mintime-1, stime), 0:n.time, xout=ptimes,
               method='constant', f=0, rule=2)$y
indx1[[i]] <- ifelse(temp1==0, 1, 1+ who[pmax(1,temp1)])
# Why not just "who[temp1]" instead of who[pmax(1,temp1)] in the
# line just above? When temp1 has zeros, the first expression
# gives a vector that is shorter than temp1, and the ifelse
# doesn't work right due to mismatched lengths.
n.event[[i]] <- cfun(temp1+1, fit$n.event[who])

if (!is.null(fit$n.censor)) {
  n.censor[[i]] <- cfun(temp1+1, fit$n.censor[who])
  j <- who[n.time] #last time point in the data
  last.n <- fit$n.risk[j] - (fit$n.event[j]+ fit$n.censor[j])
}
else {
# this is for the older survfit objects, which don't contain
# n.censor. In this case, we don't know how many of the
# people at the last time are censored then & how many go
# on further. Assume we lose them all. Note normally
# extend=FALSE, so this number isn't printed anyway.
last.n <- 0
}

# Compute the number at risk. If stime = 1,10, 20 and ptime=3,10,
# 12, then temp1 = 2,2,3: the nrisk looking ahead
# approx() doesn't work if stime is of length 1
if (n.time ==1) temp1 <- rep(1, length(ptimes))
else temp1 <- approx(stime, 1:n.time, xout=ptimes,
                    method='constant', f=1, rule=2)$y
n.risk[[i]] <- ifelse(ptimes>max(stime), last.n,
                    fit$n.risk[who[temp1]])
}

times <- unlist(newtimes)
n.risk <- unlist(n.risk)

```

```

n.event <- unlist(n.event)
n.enter <- unlist(n.enter) #may be NULL
n.censor<- unlist(n.censor) #may be NULL

indx1 <- unlist(indx1)
}

```

Repeat the code for `survfitms` objects. The only real difference is the preservation of `prev` and `cumhaz` instead of `surv`.

```

<survfitms-summary>+≡
summary.survfitms <- function(object, times, censored=FALSE,
                              scale=1, extend=FALSE,
                              rmean=getOption('survfit.rmean'),
                              ...) {
  fit <- object
  if (!inherits(fit, 'survfitms'))
    stop("summary.survfitms can only be used for survfitms objects")

  if (is.null(rmean)) rmean <- "none"

  # add some temps to make survmean work
  object$surv <- 1-object$prev
  if (is.matrix(object$surv))
    dimnames(object$surv) <- list(NULL, object$states)
  temp <- survmean(object, scale=scale, rmean)
  table <- temp$matrix #for inclusion in the output list
  rmean.endtime <- temp$end.time

  # The fit$prev object is usually a matrix but can be a vector
  # We calculate row indices first, and then deal
  # with the cases at the end.
  nprev <- if (is.matrix(fit$prev)) nrow(fit$prev) else length(fit$prev)
  if (is.null(fit$strata)) {
    nstrat <- 1
    stemp <- rep(1L, nprev)
    strata.names <- ""
  }
  else {
    nstrat <- length(fit$strata)
    stemp <- rep(1:nstrat, fit$strata)
    strata.names <- names(fit$strata)
  }
}

<survsum-findrows>

# Create an output structure

```

```

if (length(indx1)== length(fit$time) && all(indx1 == seq(along=fit$time))) {
  temp <- object #no change
  temp$time <- temp$time/scale
  temp$table <- table
  if (!is.null(temp$strata))
    temp$strata <- factor(stemp, levels=1:nstrat, labels=strata.names)
}
else if (missing(times)) {
  temp <- object
  temp$time <- temp$time[indx1]/scale
  temp$table <- table
  for (j in c("n.risk", "n.event", "n.censor", "n.enter",
             "prev", "std.err", "lower", "upper")) {
    zed <- temp[[j]]
    if (!is.null(zed)) {
      if (is.matrix(zed)) temp[[j]] <- zed[indx1,,drop=FALSE]
      else temp[[j]] <- zed[indx1]
    }
  }
  temp$cumhaz <- fit$cumhaz[, ,indx1,drop=FALSE]
  if (!is.null(temp$strata))
    temp$strata <- factor(stemp[indx1], levels=1:nstrat,
                        labels=strata.names)
}
else {
  temp <- list(n=object$n, time=times/scale,
             n.risk=n.risk, n.event=n.event,
             conf.int=fit$conf.int, type=fit$type, table=table)
  if (!is.null(n.censor)) temp$n.censor <- n.censor
  if (!is.null(n.enter)) temp$n.enter <- n.enter
  if (!is.null(fit$start.time)) temp$start.time <- fit$start.time

  # why the rbind? The user may have specified a time point before
  # the first event, and indx1=1 indicates that case
  # the cumhaz array can't be done with a 1-liner
  if (is.matrix(fit$prev)) {
    temp$prev <- rbind(0, fit$prev)[indx1,,drop=FALSE]
    zz <- ifelse(indx1==1, NA, indx1-1)
    temp$cumhaz <- fit$cumhaz(, ,zz, drop=FALSE)
    temp$cumhaz <- ifelse(is.na(temp$cumhaz), 0, temp$cumhaz)
    if (!is.null(fit$std.err))
      temp$std.err <- rbind(0, fit$std.err)[indx1,,drop=FALSE]
    if (!is.null(fit$lower)) {
      temp$lower <- rbind(0, fit$lower)[indx1,,drop=FALSE]
      temp$upper <- rbind(0, fit$upper)[indx1,,drop=FALSE]
    }
  }
}

```

```

    }
    else {
      temp$prev <- c(0, fit$prev[indx1])
      temp$cumhaz <- c(0, fit$cumhaz[indx1])
      if (!is.null(fit$std.err)) temp$std.err <- c(0, fit$std.err)[indx1]
      if (!is.null(fit$lower)) {
        temp$lower <- c(0, fit$lower)[indx1]
        temp$upper <- c(0, fit$upper)[indx1]
      }
    }
    if (!is.null(fit$strata)) {
      scout <- unlist(lapply(newtimes, length))
      temp$strata <- factor(rep(1:nstrat, scout), levels=1:nstrat,
                           labels=strata.names)
    }

    temp$call <- fit$call
    if (!is.null(fit$na.action)) temp$na.action <- fit$na.action

  }

  if (length(rmean.endtime)>0 && !is.na(rmean.endtime))
    temp$rmean.endtime <- rmean.endtime
  class(temp) <- "summary.survfitms"
  temp
}

```

9 Plotting survival curves

I found a problem where `plot.survfit`, `lines.survfit`, and `points.survfit` sometimes did different things. This is due to copied code that later changed in one function but not another. Since they have so much code in common, this file consolidates them in order to restore order by using common code blocks.

```

<plot.survfit>≡
plot.survfit<- function(x, conf.int, mark.time=TRUE,
                       mark=3, col=1,lty=1, lwd=1,
                       cex=1, log=FALSE,
                       xscale=1, yscale=1,
                       firstx=0, firsty=1,
                       xmax, ymin=0,
                       fun, xlab="", ylab="", xaxs='S', ...) {

  dotnames <- names(list(...))
  if (any(dotnames=='type'))

```

```

        stop("The graphical argument 'type' is not allowed")

        <plot-transform-ms>
        <plot-plot-setup1>
        <plot-common-args>
        <plot-first1>
        <plot-plot-setup2>
        <plot-functions>
        plot.surv <- TRUE
        <plot-draw>
    }

lines.survfit <- function(x, type='s',
                          mark=3, col=1, lty=1, lwd=1,
                          mark.time=TRUE, xscale=1,
                          firstx=0, firsty=1, xmax,
                          fun, conf.int=FALSE, ...) {

    <plot-transform-ms>
    <plot-common-args>

    if (is.logical(conf.int)) plot.surv <- TRUE
    else {
        temp <- pmatch(conf.int, c("both", "only", "none"))
        if (is.na(temp)) stop("invalid value for conf.int")
        if (temp=="none") conf.int <- FALSE else conf.int <- TRUE
        if (temp=="only") plot.surv <- FALSE else plot.surv <- TRUE
    }

    <plot-first2>
    <plot-functions>
    <plot-draw>
}

points.survfit <- function(x, xscale=1,
                          xmax, fun, ...) {

    <plot-transform-ms>
    <plot-common-args>
    if (ncol(ssurv)==1) points(stime, ssurv, ...)
    else matpoints(stime, ssurv, ...)
}

```

Block of code to transform components of a `survfitms` object so that the standard plotting methods work.

```

<plot-transform-ms>≡
if (inherits(x, "survfitms")) {
    x$surv <- 1- x$prev
}

```

```

    if (is.matrix(x$surv)) dimnames(x$surv) <- list(NULL, x$states)
    if (!is.null(x$lower)) {
      x$lower <- 1- x$lower
      x$upper <- 1- x$upper
    }
    if (missing(fun)) fun <- "event"
  }
}

<plot-common-args>≡
ssurv <- as.matrix(x$surv)
stime <- x$time
if( !is.null(x$upper)) {
  supper <- as.matrix(x$upper)
  slower <- as.matrix(x$lower)
}
else conf.int=FALSE

# Two cases where we don't put marks at the censoring times
if (inherits(x, 'survexp') || inherits(x, 'survfit.coxph')) {
  if (missing(mark.time)) mark.time <- FALSE
}

# 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)

```

The `xmax` argument is used to prune back the survival curve to a small set of time points. This is a bit of bother since we have to do our own clipping of the data to prevent warning messages from the underlying plot routines.

```

<plot-common-args>+≡
if (!missing(xmax) && any(x$time>xmax)) {
  # prune back the survival curves
  # I need to replace x's over the limit with xmax, and y's over the
  # limit with either the prior y value or firsty
  keepx <- keepy <- NULL # lines to keep
  yzero <- NULL # if all points on a curve are < xmax
  tempn <- table(stemp)
  offset <- cumsum(c(0, tempn))
  for (i in 1:nstrat) {
    ttime <-stime[stemp==i]
  }
}

```

```

    if (all(ttime <= xmax)) {
      keepx <- c(keepx, 1:tempn[i] + offset[i])
      keepy <- c(keepy, 1:tempn[i] + offset[i])
    }
    else {
      bad <- min((1:tempn[i])[ttime>xmax])
      if (bad==1) {
        keepy <- c(keepy, 1+offset[i])
        yzero <- c(yzero, 1+offset[i])
      }
      else keepy<- c(keepy, c(1:(bad-1), bad-1) + offset[i])
      keepx <- c(keepx, (1:bad)+offset[i])
      stime[bad+offset[i]] <- xmax
      x$n.event[bad+offset[i]] <- 1 #don't plot a tick mark
    }
  }
}

# ok, now actually prune it
stime <- stime[keepx]
stemp <- stemp[keepx]
x$n.event <- x$n.event[keepx]
if (!is.null(x$n.censor)) x$n.censor <- x$n.censor[keepx]

if (length(yzero))
  ssurv[yzero,] <- firsty
ssurv <- ssurv[keepy,,drop=FALSE]
if (!is.null(supper)) {
  if (length(yzero)) supper[yzero,] <- slower[yzero,] <- firsty
  supper <- supper[keepy,,drop=FALSE]
  slower <- slower[keepy,,drop=FALSE]
}
}

stime <- stime/xscale #scaling is deferred until xmax processing is done

if (!missing(fun)) {
  if (is.character(fun)) {
    tfun <- switch(fun,
      'log' = function(x) x,
      'event'=function(x) 1-x,
      'cumhaz'=function(x) -log(x),
      'cloglog'=function(x) log(-log(x)),
      'pct' = function(x) x*100,
      'logpct'= function(x) 100*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)
}
firsty <- tfun(firsty)
}

```

The plot and points routines need to process the `firstx` and `firsty` arguments since their lines start there, but the points routine does not use these. For counting process data we default to the minimal time value, and for other the default is 0. The other issue is that if log scale has been chosen we want to use a different default. However, the lines routine has no way of knowing if this is the case. Expected curves have a NULL value for `type`.

```

<plot-first1>≡
if (missing(firsty) && (is.null(x$type) ||
                        x$type=="mright" || x$type=="mcounting")) {
  firsty <- NA
  firstx <- min(stime)
}
else if (missing(firstx)) {
  if (!is.null(x$start.time))
    firstx <- x$start.time
  else {
    if (logx || (!missing(fun) && is.character(fun) && fun=='cloglog'))
      firstx <- min(stime[stime>0])
    else firstx <- min(0, stime)
  }
}

```

```

#The default for plot is to add confidence limits if there is only one curve
if (missing(conf.int)) conf.int <- (ncurve==1)

```

```

<plot-setup-marks>

```

The dumbed down version for lines, which won't know about log.

```

<plot-first2>≡
if (missing(firsty) && (is.null(x$type) ||
                        x$type=="mright" || x$type=="mcounting")) {
  firsty <- NA
  firstx <- min(stime)
}
else if (missing(firstx)) {
  if (!is.null(x$start.time))
    firstx <- x$start.time
  else firstx <- min(0, stime)
}

```

```

}
<plot-setup-marks>
<plot-setup-marks>≡
# Marks are not placed on confidence bands
mark <- rep(mark, 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
# 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) {
  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)
}

```

Here is the rest of the setup for the plot routine, mostly having to do with setting up axes.

```

<plot-plot-setup1>≡
if (is.logical(log)) {
  logy <- log
  logx <- FALSE
  if (logy) logax <- 'y'
  else      logax <- ""
}
else {
  logy <- (log=='y' || log=='xy')
  logx <- (log=='x' || log=='xy')
  logax <- log

```

```

}

# The special x axis style only applies when firstx is not given
if (missing(xaxs) && (firstx!=0 || !missing(fun) ||
    (missing(fun) && inherits(x, "survfitms"))))
  xaxs <- par("xaxs") #use the default

<plot-plot-setup2>≡
#axis setting parmaters that depend on the fun argument
if (!missing(fun)) {
  if (is.character(fun)) {
    if (fun=='log' || fun=='logpct') logy <- TRUE
    if (fun=='cloglog') {
      logx <- TRUE
      if (logy) logax <- 'xy'
      else logax <- 'x'
    }
  }
  ymin <- tfun(ymin) #lines routine doesn't have it
}

# Do axis range computations
if (xaxs=='S') {
  #special x- axis style for survival curves
  xaxs <- 'i' #what S thinks
  tempx <- max(stime) * 1.04
}
else tempx <- max(stime)
tempx <- c(firstx, tempx, firstx)

if (logy) {
  tempy <- range(ssurv[is.finite(ssurv)& ssurv>0])
  if (tempy[2]==1) tempy[2] <- .99
  if (any(ssurv==0)) {
    tempy[1] <- tempy[1]*.8
    ssurv[ssurv==0] <- tempy[1]
    if (!is.null(supper)) {
      supper[supper==0] <- tempy[1]
      slower[slower==0] <- tempy[1]
    }
  }
  tempy <- c(tempy, firsty)
}
else tempy <- c(range(ssurv[is.finite(ssurv)] ), firsty)

if (missing(fun)) {

```

```

    tempx <- c(tempx, firstx)
    tempy <- c(tempy, ymin)
}

#
# Draw the basic box
#
plot(tempx, tempy*yscale, type='n', log=logax,
      xlab=xlab, ylab=ylab, xaxs=xaxs,...)

if(yscale != 1) {
  if (logy) par(usr =par("usr") -c(0, 0, log10(yscale), log10(yscale)))
  else par(usr =par("usr")/c(1, 1, yscale, yscale))
}

```

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. Why didn't I use the same trick for xscale? Lack of foresight. And now there are hundreds of lines of code that have an xscale argument to `lines()` so I don't dare drop 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) {
  if (is.na(x[1] + y[1])) {
    x <- x[-1]
    y <- y[-1]
  }
  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 horizontal sequences like
    # (1, .2), (1.4, .2), (1.8, .2), (2.3, .2), (2.9, .2), (3, .1)
    # with (1, .2), (3, .1). They are slow, and can smear the looks
    # of the line type.
    dupy <- c(!duplicated(y)[-n], TRUE)
    n2 <- sum(dupy)

    #create a step function
    xrep <- rep(x[dupy], c(1, rep(2, n2-1)))
    yrep <- rep(y[dupy], c(rep(2, n2-1), 1))
    list(x=xrep, y=yrep)
  }
}

```

```

drawmark <- function(x, y, mark.time, censor, ...) {
  if (!is.numeric(mark.time)) {
    xx <- x[censor]
    yy <- y[censor]
  }
  else { #interpolate
    xx <- mark.time
    yy <- approx(x, y, xx, method="constant", f=0)
  }
  points(xx, yy, ...)
}

```

The code to actually draw curves for the plot.

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)

for (i in unique(stemp)) { #for each strata
  who <- which(stemp==i)
  xx <- c(firstx, stime[who])
  censor <- if (is.null(x$n.censor))
    (x$n.event[who] ==0) else (x$n.censor[who] >0) #places with a censoring
  for (j in 1:ncol(ssurv)) {
    yy <- c(firsty, ssurv[who,j])
    if (plot.surv) {
      lines(dostep(xx, yy), lty=lty[c2], col=col[c2], lwd=lwd[c2])
      if (mark.time) drawmark(xx, yy, mark.time, censor,
        pch=mark[c1], col=mcol[c1])
    }
    xend[c1] <- max(xx)
    yend[c1] <- yy[length(yy)]
    c1 <- c1 +1
    c2 <- c2 +1

    if (conf.int) {
      lines(dostep(xx, c(firsty, slower[who,j])), lty=lty[c2], col=col[c2],
        lwd=lwd[c2])
      c2 <- c2 +1
      lines(dostep(xx, c(firsty, supper[who,j])), lty=lty[c2], col=col[c2],
        lwd= lwd[c2])
      c2 <- c2 + 1
    }
  }
}

```

```
    }  
  }  
invisible(list(x=xend, y=yend))
```

References

- [1] M. H. Gail, J. H. Lubin, and L. V. Rubinstein. Likelihood calculations for matched case-control studies and survival studies with tied death times. *Biometrika*, 68:703–707, 1981.