# Survival Package Functions

Terry Therneau

October 27, 2010

## Contents



# 1 Introduction

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

This is the definition of a coding style called *literate programming*. I first made use of it in the *coxme* library and have become a full convert. For the survival library only selected objects are documented in this way; whenever I find need for a major revision I "convert" the source. An underlying motivation is to leave code that is well enough explained that someone else can take it over.

# 2 residuals.survreg

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.

$\langle$*rsr-resid*$\rangle\equiv$

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

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

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

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

The log-likelihood can be written as

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

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

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

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

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

⟨*rtr-deriv*⟩≡

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

```
    z2 <- 0
    }

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

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

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

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

⟨*rtr-resid2*⟩≡

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

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

else if (type=='dfbeta' || type== 'dfbetas' || type=='ldcase') {
    score <- deriv[,2] * x  # score residuals
```

```
    if (rsigma) {
        if (nstrata > 1) {
            d4 <- matrix(0., nrow=n, ncol=nstrata)
            d4[cbind(1:n, strata)] <- deriv[,4]
            score <- cbind(score, d4)
            }
        else score <- cbind(score, deriv[,4])
        }
    rr <- score %*% vv
    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
```

# 3 Predicted survival from a Cox model

## 3.1 Individual 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

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, ...) {

    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-compute⟩
    ⟨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.

⟨*survfit.coxph-setup*⟩≡
```
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")
    survtype <- match(match.arg(type, temp1), temp1)
    }
if (missing(vartype)) {
    vartype <- survtype
    }
else {
    temp2 <- c("greenwood", "tsiatis", "efron", "aalen")
    vartype <- match(match.arg(vartype, temp2), temp2)
    if (vartype=="tsiatis") vartype<- "aalen"
    }

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 model.frame is needed. (The primary user of the default case is the cph function from Frank Harrell's libraries, since `coxph` by default does not save $x$.) The "failed to reconstruct" error messages are to avoid nonsense results 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)
```

Also check for `type` of the same flavor; since `coxph` can only deal with these two types, any other must signal a change in the data. We have to use `object['x']` instead of `object$x` since the latter will pick off the `xlevels` component.

⟨*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 (is.null(object[['y']])) y <- model.response(mf)
else y <- object[['y']]

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

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(0L,n)
else strata <- mf[[temp$vars]]
```

A key variable for the computation is the risk score $\exp(X\beta)$ for each original observation along with the risk score for the target subject(s). There are three choices for the new data

- For predictions with time-dependent covariates the user will have specified the option `individual=TRUE`, and the new data set contains the covariate trace over time of a single individual. We need to retrieve the covariates, strata, and repsonse from the new data set.
- For ordinary predictions only the covariates are needed.
- If newdata is not present we assume that this is the ordinary case, and use the value of `object$means` as the default covariate set. (This is not a good idea statistically.)

Since the new data might not have all of the levels present for one of the factors, we need to ensure that levels match the original data. In all cases the computation depends only on the *difference* between the original covariates and the target set. We subtract the means from each column of the $X$ matrices to avoid any problems with overflow in the exponential function.

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. The other 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.

⟨*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)
    }
else {
    varmat <- object$var
    x <- scale(x, center=object$means, scale=FALSE)
    coef <- ifelse(is.na(object$coefficients), 0, object$coefficients)
    }
risk <- exp(x%*% coef + offset - mean(offset))

if (individual) {
    if (missing(newdata)) stop("The newdata argument must be present")
    if (!is.data.frame(newdata)) stop("Newdata must be a data frame")
```

```
    temp <- untangle.specials(Terms, 'cluster')
    if (length(temp$vars)) Terms2 <- Terms[-temp$terms]
    else Terms2 <- Terms
    mf2 <- model.frame(Terms2, newdata, xlev=object$xlevels)

    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 <- rep(0, nrow(mf2))

    x2 <- model.matrix(Terms2, mf2)[,-1, drop=FALSE]  #no intercept
    if (length(x2)==0) x2 <- matrix(0.0, nrow=nrow(mf2), ncol=1)
    else x2 <- scale(x2, center=object$means, 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")
    }
else {
    if (missing(newdata)) {
        x2 <- matrix(0.0, nrow=1, ncol=ncol(x))
        offset2 <- 0
        }
    else {
```

For backwards compatabliity: 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 element have a name. If not we attach the variable names from the model, and assume that it's the right order. (Documentation of this ability has been suppressed, however.)

⟨*survfit.coxph-setup*⟩+≡

```
        if (!is.data.frame(newdata)) {
            if (is.list(newdata)) newdata <- data.frame(newdata)
            else if (is.numeric(newdata) &&
                     length(newdata)==length(object$coefficients)) {
                if (is.null(names(newdata)))
                    names(newdata) <- names(object$coefficients)
                newdata <- data.frame(as.list(newdata))
```

```
                }
            else stop("Invalid newdata object")
            }
        Terms2 <- delete.response(Terms)
        temp <- untangle.specials(Terms2, 'cluster')
        if (length(temp$vars)) Terms2 <- Terms2[-temp$terms]
        temp <- untangle.specials(Terms2, 'strata')
        if (length(temp$vars)) Terms2 <- Terms2[-temp$terms]
        mf2 <- model.frame(Terms2, newdata, xlev=object$xlevels)
        x2 <- model.matrix(Terms2, mf2)[,-1, drop=FALSE]  #no intercept
        x2 <- scale(x2, center=object$means, scale=FALSE)
        offset2 <- model.offset(mf2)
        if (length(offset2) >0) offset2 <- offset2 - mean(offset)
        else offset2 <- 0
        }
    }
newrisk <- exp(c(x2 %*% coef) + offset2)
```

Now, we're ready to do the main compuation. 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, but all have a different number of elements in their final result. So, for each strata we generate a list, and then unpack the results at the end. 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.

⟨*survfit.coxph-compute*⟩≡

```
ustrata <- levels(as.factor(strata))
nstrata <- length(ustrata)
survlist <- vector('list', nstrata)
if (se.fit) varhaz <- 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)
    }
```

In an ordinary survival curve object with multiple strata, as produced by `survfitKM`, the time, survival and etc components are each a single vector that contains the results for strata 1, followed by strata 2, .... The strata compontent is a vector of integers, one per strata, that gives the number of elements belonging to each stratum. The reason is that each strata will have a different number of observations, so that a matrix form was not viable, and the underlying C routines were not capable of handling lists (the code predates the .Call function by a decade). The lists above will be concatonated into vectors.

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 marix of survival and variance estimates, one column for each of the distinct target values. If $\Lambda_0$ 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

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

$V$ is the variance matrix for $\beta$ 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.

⟨*survfit.coxph-compute*⟩+≡

```
if (!individual) {
    cumhaz <- unlist(lapply(survlist, function(x) x$cumhaz))
    varhaz <- unlist(lapply(survlist, function(x) cumsum(x$varhaz)))
    nevent <- unlist(lapply(survlist, function(x) x$n.event)) #weighted
    ndeath <- unlist(lapply(survlist, function(x) x$ndeath))  #unweighted
    xbar   <- t(matrix(unlist(lapply(survlist, function(x) t(x$xbar))),
                  nrow=ncol(x)))
    hazard <- unlist(lapply(survlist, function(x) x$hazard))

    if (survtype==1)
        surv <-unlist(lapply(survlist, function(x) cumprod(x$surv)))
    else surv <- exp(-cumhaz)

    if (is.matrix(x2) && nrow(x2) >1) {  #more than 1 row in newdata
        surv <- outer(surv, newrisk, '^')
```

```
            varh <- matrix(0., nrow=length(varhaz), ncol=nrow(x2))
            for (i in 1:nrow(x2)) {
                dt <- outer(cumhaz, x2[i,], '*') - xbar
                varh[,i] <- (varhaz + rowSums((dt %*% varmat)* dt)) *
                    newrisk[i]^2
                }
            }
        else {
            surv <- surv^newrisk
            dt <-  outer(cumhaz, c(x2)) - xbar
            varh <- (varhaz + rowSums((dt %*% varmat)* dt)) *
                newrisk^2
            }
```

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 quadratice 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.

⟨*survfit.coxph-compute*⟩+≡

```
        result <- list(n=as.vector(table(strata)),
                       time=unlist(lapply(survlist, function(x) x$time)),
                       n.risk= unlist(lapply(survlist, function(x) x$n.risk)),
                       n.event=nevent,
                       n.censor=unlist(lapply(survlist, function(x) x$n.censor)),
                       surv=surv,
                       type=type)
    if (nstrata >1) {
            result$strata <- unlist(lapply(survlist, function(x) length(x$n.risk)))
            names(result$strata) <- ustrata
            }
        }
```

For the case with `individual=TRUE` we always produce a single survival curve. 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 10 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, 14 and 17. The deltas saved below are 2-1, 5-2, 7-5, 3+ 14-10, 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 ≤ 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, we can't let them count the jump twice.

⟨*survfit.coxph-compute*⟩+≡
```
else {
    ntarget <- nrow(x2)  #number of different time intervals
    surv <- vector('list', ntarget)
    n.event <- n.risk <- n.censor <- varh1 <- varh2 <-  time <- surv
    stemp <- match(strata2, ustrata)
    timeforward <- 0
    for (i in 1:ntarget) {
        slist <- survlist[[stemp[i]]]
        indx <- which(slist$time > y2[i,1] & slist$time <= y2[i,2])
        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
        }
    varh <- cumsum(unlist(varh1)) + unlist(varh2)

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

    result <- 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, type=type)
    }
```

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

The risk score for each subject is $r_i = \exp(X_i\beta)$.

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

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

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

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

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

For simple survival data the risk sum $\sum w_i r_i Y_i(t)$ for all the unique death times $t$ is fast to compute as a cumulative sum, starting at the longest followup time an 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 then all the remaining calculations together.
- Use the difference of two cumulative sums, one ordered by start time and one ordered by stop time. This is $O(2n)$ for the intial sums. The problem here is potential round off error if the sums get large, 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(time=time, n.event=nevent, n.risk=irisk, n.censor=ncens,
                  hazard=haz,
                  cumhaz=cumsum(haz), varhaz=varhaz, ndeath=ndeath,
                  xbar=apply(matrix(xbar, ncol=nvar),2, cumsum))
    if (survtype==1) result$surv <- km$inc
    result
    }
```

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

⟨*agsurv4*⟩≡

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

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

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

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

**n** number of observations (unique death times)

**d** number of deaths at that time

**nvar** number of covariates

**x1** weighted number at risk at the time

**x2** sum of weights for the deaths

**xsum** matrix containing the cumulative sum of x values

**xsum2** matrix of sums, only for the deaths

On output the values are

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

  **sum1** `1/x1`

**sum2** `1/x1^2`

**xbar** `xsum/x1^2`

- d=2

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

  **sum2** (1/2) ( 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) ( 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 *n,      Sint *nvar,   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;

    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;
                    }
                }
            }
        }
    }
```

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$n.event > 0)
    if (nstrata >1) {
        temp <- rep(ustrata, each=result$strata)
        result$strata <- c(table(temp[keep]))
        }
    result <- lapply(result, kfun, keep)
    if (se.fit) varh <- kfun(varh, keep)
    }

if (se.fit) {
    result$std.err <- sqrt(varh)

    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
if (is.R()) class(result) <- c('survfit.cox', 'survfit')
else        oldClass(result) <- 'survfit.cox'
result
```

# 4 survexp

The arguments for `survexp` are

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

- for the Hakulinen method it is a vector of censoring times. This is the actual censoring time for censored subjecs, and is what the censoring time 'would have been' for each subject who died.
- for the conditional method it is the usual Surv(time, status) code
- for the Ederer method no response is 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 causes 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.

⟨*survexp-setup*⟩+≡

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

if(length(rate) == 1) {
    if (!missing(rmap))
        stop("The ratetable() call in the 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

# 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]]
        }
    }
# 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='+')
    m$formula <- as.formula(tform)
    }

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

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`. For a Cox model `R` will be a model matix whose covariates are coded in exactly the same way that variables were coded in the original Cox model. We call the model.matrix.coxph function to avoid 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 <- data.frame(eval(rcall, m))  # the variables matching the ratetable
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 (cbind(1:n,R), Y, max(Y), TRUE, ratetable)
    else temp<- survexp.cfit(cbind(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(cbind(as.numeric(X),R), Y, newtime,
                        conditional, ratetable)
else {
    temp <- survexp.cfit(cbind(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 <- structure(cbind(X, R),
        dimnames=list(row.names(m), c("group", varlist ))) ### was dimid)))
    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
```

# 5 predict.coxph

This function produces various types of predicted values from a Cox model. The arguments are

**obejct** 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

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

## 5.1 Setup

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

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

⟨*predict.coxph*⟩≡
```
predict.coxph <- function(object, newdata,
                        type=c("lp", "risk", "expected", "terms"),
                        se.fit=FALSE, na.action=na.pass,
                        terms=names(object$assign), collapse, ...) {
    ⟨pcoxph-init⟩
    ⟨pcoxph-getdata⟩
    ⟨pcoxph-expected⟩
    ⟨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")

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")
    }

#Do I have strata or a cluster statment?  Don't conflict names with the strata()
#  function
strat <- attr(Terms, 'specials')$strata
dropx <- NULL
if (length(strat)) {
    stemp <- untangle.specials(Terms, 'strata', 1)
    dropx <- stemp$terms
    }
if (length(attr(Terms, 'specials')$cluster)) {
    temp <- untangle.specials(Terms, 'cluster', 1)
    dropx <- c(dropx, temp$terms)
    }
if (length(dropx)) Terms2 <- Terms[-dropx]
else  Terms2 <- Terms
if (type != 'expected') Terms2 <- delete.response(Terms2)

na.action.used <- object$na.action
has.offset <- !is.null(attr(Terms, 'offset'))
```

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

- for type='expected' residuals we need the orignal survival y. This is saved in coxph objects by default so will only need to be fetched in the highly unusual case that a user specfied y=FALSE in the orignal call.

- for any call with either newdata, standard errors, 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 any

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

⟨*pcoxph-getdata*⟩≡
```
if (type == 'expected') {
    y <- object[['y']]
    if (is.null(y)) {  # very rare case
        mf <- model.frame(object)
        y <-  model.extract(mf, 'response')
        }
    }

if (se.fit || !missing(newdata) || type=='terms' || length(strat)) {
    need.x <- TRUE
    if (is.null(object[['x']])) {
        mf <- model.frame(object)
        x <- model.matrix(delete.response(Terms2), mf,
                    contr=object$contrasts)[,-1,drop=FALSE]
        if (length(strat)) {
            if (length(stemp$vars)==1) oldstrat <- mf[[stemp$vars]]
            else oldstrat <- strata(mf[,stemp$vars], shortlabel=TRUE)
            }
        else oldstrat <- rep(0L, nrow(mf))
        weights <- model.weights(mf)
        offset <- model.offset(mf)
        }
    else {
        x <- object[['x']]
        oldstrat <- object$strata
        weights <- object$weights
        offset <-  object$offset
        }
    if (is.null(weights)) weights <- rep(1.0, nrow(x))
    if (is.null(offset))  offset  <- rep(0.0, nrow(x))
    if (length(oldstrat)==0) oldstrat <- rep(0L, nrow(x))
    }
else {
    oldstrat <- rep(0L, n)
    offset <- 0
```

```
        need.x <- FALSE
        }

    if (!missing(newdata)) {
        mf <- model.frame(Terms2, data=newdata, xlev=object$xlevels,
                          na.action=na.action)
        newx <- model.matrix(Terms2, mf,
                             contr=object$contrasts)[,-1,drop=FALSE]
        if (length(strat)) {
            if (length(stemp$vars)==1) newstrat <- mf[[stemp$vars]]
            else newstrat <- strata(mf[,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 <- rep(0L, nrow(mf))

        newoffset <- model.offset(mf)
        if (is.null(newoffset)) newoffset <- rep(0.0, nrow(newx))
        na.action.used <- attr(mf, 'na.action')
        if (type== 'expected') {
            newy <- model.response(mf)
            if (attr(newy, 'type') != attr(y, 'type'))
                stop("New data has a different survival type than the model")
            }
        }
```

## 5.2 Expected hazard

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 (type=='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 compuation below, whenever operating on the original data set. This is a consequence of the

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

⟨*pcoxph-expected2*⟩≡
```
ustrata <- unique(oldstrat)
risk <- exp(object$linear.predictors)
x <- scale(x, center=object$means, scale=FALSE)
if (se.fit) {
    se <- double(nrow(mf))
    }
if (missing(newdata))
    se <- double(nrow(mf))
if (!missing(newdata)) {
    se <- double(nrow(mf))
    pred <- se
    newx <- scale(newx, center=object$means, scale=FALSE)
    newrisk <- c(exp(newx %*% object$coef))
    }
survtype= ifelse(fit$method=='efron', 3,2)
for (i in ustrata) {
    indx <- which(oldstrat == i)
    afit <- agsurv(y[indx,,drop=F], x[indx,,drop=F],
                              weights[indx], risk[indx],
                              survtype, survtype)
    afit.n <- length(afit$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
    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]
            }
```

```
            }
        }
    }
```

## 5.3 Linear predictor, risk, and terms

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 $3\beta$. The relative predictor

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

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

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

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

Start with the simple ones

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

if (missing(newdata)) {
    offset <- offset - mean(offset)
    if (length(strat)) {
        for (i in unique(oldstrat)) {
```

```
            j <- which(oldstrat==i)
            if (length(j)==1) x[j,] <- 0   # only 1 subject in the strata
            else if (length(j) >1) {
                xmean <- colSums(x[j,] * weights[j]) / sum(weights[j])
                x[j,]  <- scale(x[j,], center=xmean, scale=FALSE)
                }
            }
        newx <- x
        }
    else if (need.x) newx <- scale(x, center=object$means, scale=FALSE)
    }
else {
    offset <- newoffset - mean(offset)
    if (length(strat)) {
        for (i in unique(oldstrat)) {
            j <- which(newstrat==i)    #no matches is a possibility
            if (length(j) ==1) newx[j,] <- 0
            else if (length(j) >1) {
                xmean <- colSums(x[j,,drop=FALSE] * weights[j]) / sum(weights[j])
                newx[j,]  <- scale(newx[j,], center=xmean, scale=FALSE)
                }
            }
        }
    else newx <- scale(newx, center=object$means, scale=FALSE)
    }

if (type=='lp' || type=='risk') {
    if (need.x) pred <- 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 $X$, $\beta$ and $V$ (the variance matrix) to that subset, then the linear predictor is $X\beta$ with variance matrix $XVX'$. The standard errors are the square root of the diagonal of this latter matrix. This can be computed, as colSums((X

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

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

    for (i in 1:nterms) {
        tt <- asgn[[i]]
        tt <- tt[!is.na(object$coefficients[tt])]
        xtt <- newx[,tt, drop=F]
        pred[,i] <- xtt %*% object$coefficient[tt]
        if (se.fit)
            se[,i] <- sqrt(rowSums((xtt %*% object$var[tt,tt]) *xtt))
        }
    pred <- pred[,terms, drop=F]
    if (se.fit) se <- se[,terms, drop=F]

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

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

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

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

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