Survival Analysis in XLISP-Stat
A semiliterate program.

Thomas Lumley
Department of Biostatistics
University of Washington

September 9, 1998

Contents

1 Outline

This document contains program code and examples of survival analyses in XLISP-Stat, structured using the `noweb`(Ramsey, 1993) literate programming system. It is described as a "semiliterate" program because most of the code was already written before it was converted to use `noweb`.

The structure of the program code is

2a ⟨*program* 2a⟩≡
```
⟨model formula object and methods 81⟩
(provide "modelformula")
⟨support functions 57⟩
⟨cox regression object and methods 47⟩
(provide "coxreg")
⟨user-level functions 4⟩
⟨default options 6⟩
```

The detailed examples are based on two data sets used by Fleming & Harrington (1991) and cover analyses and diagnostics in that book, diagnostics due to Grambsch & Therneau (1994), and recurrent event analyses described by Therneau & Hamilton (1997). The best way to use the examples is to cut and paste them into an XLISP-Stat session, so you can see both input and output. This is easiest under Unix when using Emacs with ESS or inferior-lisp-mode. If the file is just loaded into XLISP-Stat the code does not appear on the screen, making the output somewhat confusing.

2b ⟨*examples* 2b⟩≡
```
(require "coxreg")
⟨eg:fitting models 7⟩
⟨eg:model summaries 17⟩
⟨eg:expected survival examples 20⟩
⟨eg:diagnostic examples 29⟩
⟨eg:tie examples 36⟩
⟨eg:basic examples 39⟩
⟨eg:time-dependence 40⟩
⟨eg:recurrent events 43⟩
```

To help new users of XLISP-Stat who are familiar with Cox regression the `coxexample.lsp` file contains a simple analysis of the data from Fleming & Harrington (1991). This only uses a few of the available survival analysis features, so you should follow up by reading the Examples sections of this document for the other features.

3 ⟨*introductory example* 3⟩≡

```
(require "coxreg")
(load "pbc.lsp")
(cox-model :times time :status status :x bili)
(cox-model :times time :status status :x (log bili))

(def mayo-model-formula
     (as-formula '((term age) (term (log bili)) (term (log protime))
                   (term (log alb)) (term edtrt))))

(def mayo-model (cox-model :times time :status status :x mayo-model-formula))

(send mayo-model :plot-delta-betas)

(def typical (list 0 (log 3.5) (log 10) (log 1.75 ) 50))
(send mayo-model :plot-survival :xvec typical)
```

2 Analysis and Modelling of Time-to-Event Data

2.1 Notation

Our mathematical notation follows that of Fleming & Harrington (1991). We define for individual i the right-continous counting process $N_i(t)$ that counts the number of failures for subject i at or before time t, and the left-continuous 'at risk' process $Y_i(t)$, which is 1 when the subject is under observation and 0 otherwise. Each individual also has a possibly time-dependent vector of covariates $Z_i(t)$.

For standard survival analysis there is at most one event for each individual, and individuals are under observation from their entry time until they fail, are lost to followup, or until the study ends. We denote by X_i the last time at which individual i is under observation, which is either the failure time or the censoring time. We write Δ_i for the failure indicator $\Delta_i \equiv dN_i(X_i)$, which is 1 for a failure and 0 for a censoring. If individuals are not under observation starting from time 0 but from some later time this is called 'left truncation'.

The cumulative hazard for individual i, $\Lambda_i(t)$, gives the expected number of events for the individual by time t. In fact, it has the stronger property that for any times $t > s$, $E[N_i(t) - \Lambda_i(t)] = 0$ conditional on all information up to time s. The difference

$$M_i(t) = N_i(t) - \Lambda_i(t)$$

is called the counting process martingale. The number of events, number of subjects under observation and expected numbers of events for the whole sample are denoted $N(t)$, $Y(t)$ and $\Lambda(t)$.

The Cox (proportional hazards) regression model is that

$$d\Lambda_i(t) = d\Lambda_0(t) e^{\beta' Z_i(t)} Y_i(t)$$

for some regression coefficients β and unspecified baseline cumulative hazard $\Lambda_0(t)$. This is also called the Andersen–Gill multiplicative intensity model, especially in the case when more than one event per subject is possible. The model is fitted by maximising the partial likelihood, the product over all event times t of

$$\frac{\exp(\beta' Z_i(t))}{\sum_{j=1}^{n} Y_j(t) \exp(\beta' Z_j(t))}$$

where individual i is the one failing at time t. This is the conditional probability that individual i is the one who fails, given that there is exactly one failure.

The log partial likelihood is a function of weighted moments of the covariates $Z_i(t)$ over subjects still under observation, weighted by the hazard ratio $\exp(\beta' Z_i(t))$. These are written

$$\begin{aligned}
S^{(0)}(t,\beta) &= \frac{1}{n} \sum_{i=1}^{n} Y_i(t) \exp(\beta' Z_i(t)) \\
S^{(1)}(t,\beta) &= \frac{1}{n} \sum_{i=1}^{n} Y_i(t) Z_i(t) \exp(\beta' Z_i(t)) \\
S^{(2)}(t,\beta) &= \frac{1}{n} \sum_{i=1}^{n} Y_i(t) Z_i(t) Z_i(t)' \exp(\beta' Z_i(t)).
\end{aligned}$$

2.2 Fitting Cox models to data

Functions intended to be called by the user define cox model objects and perform some one and two sample summaries

4 ⟨*user-level functions* 4⟩≡
⟨*cox-model* 5⟩
⟨*one and two sample functions* 37⟩

The main function from the point of view of the user is `(cox-model)`, which constructs a object of class `cox-regression-proto`.

5 ⟨*cox-model* 5⟩≡
```
(defun cox-model (&key x times status
                       (id (iseq (length times)) has-id)
                       (entry nil)
                       (strata (* 0 status) has-strata)
                       (agnostic nil)
                       (verbose *cox-verbose*)
                       (print t)
                       (init-beta nil)
                       (count-limit *cox-count-limit*)
                       (weights nil)
                       (predictor-names nil has-names)
                       (response-name "Failure time" has-response-name)
                       (ties *cox-default-ties*)
                       (suppress-warnings nil)
                       time-x
                       time-funs
                       )
"Constructs a Cox regression modelling with right censoring and
optionally left truncation and strata."
(unless (and x times status)
        (error "You must supply :x :times and :status.~%"))
⟨internals of cox-model 63⟩
)
```

The `:x`, `:times`, and `:status` arguments are required. `:x` specifies the design matrix for the model. This may be a matrix, a list of sequences giving the columns of the matrix, or a single sequence for a single-variable model. It may also be a model formula object as created by `(as-formula)` (see Section 4). The `:times` argument gives the sequence of observation (survival) times and the `:status` argument gives a sequence containing 1 for an event and 0 for a censoring. If there is left-truncation in the data, entry times can be specified in the `:entry` argument.

A stratified model can be fitted by specifying a sequence of stratum identifiers in `:strata`, which must be all numbers or all strings. A case identifier may be specified in the `:id` argument. This affects the computation of diagnostics and the model-agnostic covariance matrix but not the parameter estimates. Model-agnostic covariance matrices specified by `:agnostic t` are discussed in section 2.9 and alternative approximations for tied failure times, specified by `:ties`, in section 2.6.

Global options, stored in variables with names of the form `*cox--...*`, control a number of default settings.

6 ⟨*default options* 6⟩≡

```
(def *cox-count-limit* 20)
(def *cox-default-ties* efron-ties)
(def *cox-verbose* t)
(def *cox-tolerance* (sqrt (sqrt machine-epsilon)))
(def *cox-display-block-only* nil)
(def *cox-paranoia* t)
```

`*cox-display-block-only*` controls the display of factor variables when a model formula is used (see Section 4) and `*cox-verbose*` governs the display of iteration information. The fitting procedure (Section 3.3) terminates by default after `*cox-count-limit*` iterations or when the loglikelihood or the score vector changes by less than `*cox-tolerance*`. The default approximation for tied failure times (section 2.6) is given by `*cox-default-ties*`. The internal workings of `(cox-model)` are mostly concerned with restructuring the data to suit the fitting procedure are are dealt with in Section 3.3. The Newton–Raphson code incorporates step-halving to handle numerical instabilities in fitting. When `*cox-paranoia*` is `t`, step-halving is invoked if the fitted hazard ratio between the highest and lowest risk cases is more than `(sqrt machine-epsilon)`, otherwise it is only invoked if the log-likelihood decreases during the fitting procedure.

2.2.1 Examples

The following examples show how to fit a number of models to the Mayo Clinic primary biliary cirrhosis data from Fleming & Harrington (1991), submitted to `Statlib` by Terry Therneau. These example models will be used later in this document to illustrate diagnostics.

The Mayo model is possibly the best known predictive model fitted to survival data. It is based on data from a randomised trial of D-penicillamine to treat primary biliary cirrhosis, which is a rare, immune-mediated liver disease. The treatment appeared to have no effect, but the data from 318 patients entered in the trial allowed a predictive model to be constructed to aid doctors and patients in making future treatment decisions. The model has since been extensively validated against other patient groups and was recently used to give an estimate of the survival advantage gained from liver transplantation, the only effective treatment for this disease. The data and a description of the model selection are given in Fleming & Harrington (1991).

Here we fit and summarize the Mayo model with and without a model formula. Another predictor available in this data set is histologic stage of the disease. As `stage-model` shows it is highly predictive of survival. We do not want `stage` in the model if possible, as it can only be measured by liver biopsy, so we see if it adds anything to the prediction from the Mayo model, using both the automatically provided Wald test and the likelihood ratio test. These show that `stage` has no additional predictive value.

We will see later that there are some doubts about the proportional hazards assumptions for the variable `edtrt`, measuring level of edema (fluid retention). One possible remedy is to stratify on this variable, done in the last example in this section.

7 ⟨*eg:fitting models* 7⟩≡

```
(load "pbc.lsp")
(wait)
(def mayo-model-formula
     (as-formula '((term age) (term (log bili)) (term (log protime))
                   (term (log alb)) (term edtrt))))

(def mayo-model (cox-model :times time :status status :x mayo-model-formula))
(wait)
(def mayo-model-without-formula
     (cox-model :times time :status status
                :x (list age (log bili) (log protime) (log alb) edtrt)))

(wait)
(def stage-model (cox-model :times time :status status
                            :x (as-formula '((factor stage)))))
```

```
(wait)
(def mayo-model-stage
     (cox-model :times time :status status
                :x (as-formula '((term age) (term (log bili))
                                 (term (log protime)) (term (log alb))
                                 (term edtrt) (factor stage)
                                 ))))

(- 1 (chisq-cdf
      (* 2 (- (send mayo-model-stage :loglike)
              (send mayo-model :loglike)))
      3
      )
   )

(def mayo-model-stratified
     (cox-model :times time :status status :strata edtrt
                :x (as-formula '((term age) (term (log bili))
                                 (term (log protime)) (term (log alb))
                                 ))))
```

2.3 Model Summaries

The `:display` and `:display-with-formula` methods give coefficients, standard errors, loglikelihood and some other fitting information for models specified by a design matrix or a model formula object respectively. `:display-with-formula` will also provide Wald tests for each term in the model formula, a feature that is particularly useful for factor variables and interactions. These display methods are automatically invoked when a model is constructed by `(cox-model)`. The display can be suppressed by adding the argument `:print nil`.

The `:conf-interval` method returns a matrix containing predictor names, hazard ratios and a confidence interval for the hazard ratio. The coverage level is specified by the `:coverage` argument, with 0.95 being the default.

There are three sets of methods to provide the estimated cumulative hazard and survival function for a covariate vector specified by the `:xvec` argument (0 by default). `:survival-function` and `:cumulative-hazard-function` return a list, each element of which contains the times and function values for one stratum. `:survival-function-closure` and `:cumulative-hazard-function-closure` require a stratum argument if the model is stratified and return a function (actually a LISP function closure) that maps times to function values.

To graph these functions use `:plot-survival` and `:plot-cumulative-hazard`. In a stratified model the curves for each stratum will be plotted together but can be put on separate plots with the argument `:one-plot nil`. The `:plot-survival` method has an argument `:inverted` which can be used to draw the distribution function rather than the survival function. These 'mortality plots' are often used in clinical trials where censoring fraction is high.

The baseline survival function can be estimated in two asymptotically equivalent ways. The default method here is an extension of the Kaplan–Meier estimator

$$\hat{S}_0(t) = \prod_{t_i<t} \left(1 - \frac{dN(t_i)}{\sum_{j=1}^{n} Y_j(t_i)\exp(\beta' z_j(t_i))}\right).$$

An alternative is to estimate the survival function by transforming the cumulative hazard estimator

$$\begin{aligned} \hat{\Lambda}_0(t) &= \sum_{t_i<t} \frac{dN(t_i)}{\sum_{j=1}^{n} Y_j(t_i)\exp(\beta' z_j(t_i))} \\ \hat{S}_0(t) &= \exp\left(-\Lambda_0(t)\right). \end{aligned}$$

The latter estimator can be specified by the argument `:product-limit nil` to any of the survival-function methods.

Finally `:slider-plot` allows the effect of covariates on survival to be examined dynamically. It constructs a survival plot and a dialog box with sliders to control covariate values. The survival plot is updated as the covariates change. The code for making sliders is originally due to B. Narasimhan.

9 ⟨*model summary methods* 9⟩≡

```
(defmeth cox-regression-proto :display ()
"Prints the model summary"
   (if (send self :needs-computing) (send self :compute))
  (let ((coefs (coerce (send self :beta) 'list))
        (se-s (if (send self :agnostic)
                  (send self :agnostic-coef-standard-errors)
                (send self :coef-standard-errors)))
        (x (send self :x))
```

```
        (p-names (send self :predictor-names)))
    (format t (send self :comments))
    (if (> (length (unique (send self :strata))) 1)
        (format t "~% Stratified ")
      (format t "~%"))
    (format t "Maximum Partial Likelihood Estimates:~2%")
    (format t "~25tCoefficient~40t Std Error~%")
    (when (send self :intercept)
          (format t "Constant~25t~13,6g~40t(~,6g)~%" (car coefs) (car se-s))
          (setf coefs (cdr coefs))
          (setf se-s (cdr se-s)))
    (dotimes (i (length coefs))
             (format t "~a~25t~13,6g~40t(~,6g)~%"
                     (select p-names i) (car coefs) (car se-s))
             (setf coefs (cdr coefs) se-s (cdr se-s)))
    (format t "~%")
    (format t "-2log-likelihood:~25t~13,6g~%" (* -2 (send self :loglike)))
    (format t "Number of cases:~25t~9d~%" (send self :num-cases))
    (format t "Number of records: ~25t~9d~%" (length (send self :keyint)))
    (format t "Tie handling: ~25t")
    (if (> (sum (send self :tieint)) 0)
        (send (send self :tie-method) :prettyprint)
      (format t "No ties present~%"))
 ))

(defmeth cox-regression-proto :display-with-formula
  (&key (block-only *cox-display-block-only*))
"Arguments: &key (block-only *cox-display-block-only*)
Prints the model summary based on a model formula"
(cond ( (null (send self :formula)) (error "No formula present"))
      (t
       (if (send self :needs-computing) (send self :compute))
       (let* ((beta (coerce (send self :beta) 'list))
              (formula (send self :formula))
              (cov-mat (if (send self :agnostic)
                           (send self :agnostic-covariance-matrix)
                         (send self :covariance-matrix)))
              (nblocks (length (send formula :block-indices)))
              (block-indices (send formula :block-indices))
              (varnames (send self :predictor-names))
              )
         (format t (send self :comments))
         (if (> (length (unique (send self :strata))) 1)
             (format t "~% Stratified ")
           (format t "~%"))
         (format t "Maximum Partial Likelihood Estimates:~2%")
```

```
          (format t "~a~20t~a~35t~a~%" "Block" "Wald Chisq" "p-value")
          (dolist (i (iseq 0 (- nblocks 1)))
                  (block-test (elt block-indices i)
                              beta
                              cov-mat
                              :names varnames
                              :blockname  (elt (send formula :block-names) i)
                              :block-only block-only))
          (format t "~%")
          (format t "-2log-likelihood:~25t~13,6g~%"
                  (* -2 (send self :loglike)))
          (format t "Number of cases:~25t~9d~%"
                  (send self :num-cases))
          (format t "Number of records: ~25t~9d~%"
                  (length (send self :keyint)))
          (format t "Tie handling: ~25t")
          (if (> (sum (send self :tieint)) 0)
              (send (send self :tie-method) :prettyprint)
            (format t "No ties present~%"))
          )
        )
       )
)

(defmeth cox-regression-proto :conf-interval
  (&key (coverage 0.95) (names t))
"Arguments: &key  (coverage 0.95) (names t)
Produces a matrix of Wald confidence intervals for beta with predictor
names if requested  and  transforms them to the hazard ratio scale"
(let* ((beta (send self :beta))
       (se (sqrt (diagonal (send self :covariance-matrix))))
       (zalpha (normal-quant (+ coverage (/ (- 1 coverage) 2))))
       (baseci (apply #'bind-columns (list beta
                                           (- beta (* zalpha se))
                                           (+ beta (* zalpha se)))))
       (xform (exp baseci))
       (name-list (send self :predictor-names))
       )
  (if names (bind-columns name-list xform) xform)
  )
)

(defmeth cox-regression-proto :cumulative-hazard-function (&key (xvec nil))
"Cumulative hazards evaluated at given covariate vector"
(let* (
       (hazards (send self :cumulative-hazard))
```

```
        (beta (send self :beta))
        (rr (exp (if xvec (inner-product xvec beta) 0)))
        )
   (map-elements #'(lambda (stratum) (list (first stratum)
                                           (second stratum)
                                           (* rr (third stratum))))
                 hazards)
   )
 )

 (defmeth cox-regression-proto :survival-function
   (&key (xvec nil) (product-limit t))
 "Args: &key (xvec nil) (product-limit t)
 Kaplan-Meier or near offer
 Evaluated at specified covariate vector"
 (let* (
        (hazards (send self :cumulative-hazard-function ))
        (beta (send self :beta))
        (rr (exp (if xvec (inner-product xvec beta) 0)))
        )
   (if product-limit
       (map-elements
        #'(lambda (stratum)
            (let* (
                   (dcumhaz (difference (third stratum)))
                   (survival (^ (append-seq #(1) (cumprod (- 1 dcumhaz))) rr))
                   )
              (list (first stratum) (second stratum) survival)
              )
            )
        hazards
        )
     (map-elements
      #'(lambda (stratum)
          (list (first stratum) (second stratum)
                (exp (- (* rr (third stratum))))))
      hazards
      )
     )
   )
 )

 (defmeth cox-regression-proto :survival-function-closure
   (&key (stratum 0 has-stratum) (xvec nil) (product-limit t))
 "Arguments: &key (stratum 0 has-stratum)  (xvec nil) (product-limit t)
 Function closure giving survival function for specified stratum and
```

```
  optionally covariate pattern"
  (unless (or has-stratum (= 1 (length (unique (send self :strata)))))
          (error "Must specify stratum"))
  (let* (
         (kms (assoc stratum (coerce (send self :survival-function
                                           :xvec xvec
                                           :product-limit product-limit)
                                     'list)
                     :test #'equalp
                     ))
         (ff (if kms  #'(lambda (ti) (interpolate ti (second kms) (third kms)))
               (error "Stratum ~a not found" stratum)))
         )
    ff
    )
  )


  (defmeth cox-regression-proto :cumulative-hazard-function-closure
    (&key (stratum 0 has-stratum)  (xvec nil))
  "Arguments: &key (stratum 0 has-stratum)  (xvec nil)
  Function closure giving cumulative hazard function for specified
  stratum and optionally covariate pattern"
  (unless (or has-stratum (= 1 (length (unique (send self :strata)))))
          (error "Must specify stratum"))
  (let* (
         (haz (assoc stratum
                     (coerce (send self :cumulative-hazard-function
                                   :xvec xvec)
                             'list)
                     :test #'equalp
                     ))
         (ff (if haz
                 #'(lambda (ti) (interpolate ti (second haz) (third haz)))
               (error "Stratum ~a not found" stratum)))
         )
    ff
    )
  )

  (defmeth cox-regression-proto :plot-cumulative-hazard
    (&key (one-plot t)(xvec nil))
  "Arguments: (&key (one-plot t) (xvec nil)
  Plots cumulative hazard functions, for baseline or for specified
  covariate pattern"
  (let* (
```

```
        (hazards (send self :cumulative-hazard-function :xvec xvec))
        )
  (cond ((and one-plot (> (length hazards) 1))
         (let* ((firstone (car hazards))
                (rest (cdr hazards))
                (plot (plot-steps (second firstone) (third firstone)
                                  :variable-labels (list "time" "Hazard")))
                )
           (map-elements #'(lambda (stratum)
                     (add-steps (second stratum) (third stratum) plot)) rest)
           (send plot :adjust-to-data)
           (list plot)
           ))
         (t (map-elements #'(lambda (stratum)
                    (plot-steps (second stratum) (third stratum)
                                :title (format nil "Stratum ~a"
                                               (first stratum))
                                :variable-labels (list "time" "Hazard")))
                          hazards)
           )
         )
  )
)

(defmeth cox-regression-proto :plot-survival
  (&key (inverted nil) (one-plot t) (xvec nil) (product-limit t))
"Arguments: &key (inverted nil) (one-plot t) (xvec nil) (product-limit t)
Plots survival function (or mortality function)
for each stratum at baseline or specified covariate pattern."
(let* (
       (survivals (send self :survival-function
                        :xvec xvec
                        :product-limit product-limit))
       )
  (cond ((and one-plot (> (length survivals) 1))
         (let* ((firstone (car survivals))
                (rest (cdr survivals))
                (plot (plot-steps (second firstone)
                                  (if inverted
                                      (- 1 (third firstone))
                                    (third firstone))
                                  :variable-labels
                                  (list "time"
                                        (if inverted
                                            "Proportion failing"
                                          "Proportion surviving"))))
```

```
                )
            (map-elements #'(lambda (stratum)
                              (add-steps (second stratum)
                                   (if inverted
                                       (- 1 (third stratum))
                                     (third stratum))
                                   plot))
                          rest)
            (send plot :adjust-to-data)
            (if (null inverted) (send plot :range 1 0 1))
            (list plot)
            ))
          (t
           (let (
                 (plots (map-elements
                         #'(lambda (stratum)
                             (plot-steps (second stratum)
                                         (if inverted
                                             (- 1 (third stratum))
                                           (third stratum))
                                         :title (format nil
                                                        "Stratum ~a"
                                                        (first stratum))
                                         :variable-labels
                                         (list "time" (if inverted
                                                          "Proportion failing"
                                                        "Proportion surviving"))))
                         survivals))
                 )
            (unless inverted
                    (map-elements
                     #'(lambda (plot) (send plot :range 1 0 1))
                     plots))
            plots
            )
           )
          )
    )
  )

  (require "sliders")

  (defmeth cox-regression-proto :slider-plot ()
  "Arguments:
  Draws a survival plot with sliders"
  (let* (
```

```
        (x (column-list (send self :x)))
        (names (send self :predictor-names))
        (ranges (mapcar #'(lambda (xi)
                            (if (< (length (unique xi)) 10)
                                (sort-data (unique xi))
                                (list (min xi) (max xi)))
                          )
                x))
        (current-xvec (mapcar #'min x))
        (current-plot (elt (send self :plot-survival :one-plot t
                                 :xvec current-xvec) 0))
        (actions (mapcar
                  #'(lambda (i)
                      (format nil
                              "(lambda (x) (adjust-to-sliders x ~s))" i))
                  (iseq (length x))))
        (lspec (mapcar #'(lambda (i) (list (elt names i)
                                           (elt ranges i)
                                           (elt actions i)))
                       (iseq (length x))))
        )
    (defun adjust-to-sliders (x i)
      (setf (select current-xvec i) x)
      (send current-plot :clear-points :draw nil)
      (send current-plot :clear-lines :draw nil)
      (map-elements #'(lambda (stratum)
                        (add-steps (second stratum)
                                   (third stratum)
                                   current-plot))
                    (send self :survival-function :xvec current-xvec) )
      (send current-plot :range 1 0 1)
      )
    (make-and-run-sliders lspec)
    )
  )
```

2.3.1 Examples

We have already seen the model summaries given automatically when a model is fitted. These are produced and can be reproduced by sending the `:display` or `:display-with-formula` messages.

The result of the `:conf-interval` method is a matrix and so is best displayed with `(print-matrix)` to make the columns line up. The columns are the hazard ratio estimate $\exp(\hat{\beta})$ and a 95% confidence interval.

The baseline survival function returned by `:survival-function` is the estimated survival function for an individual with zero values for all the covariates. This is frequently unreasonable and we see for the Mayo model that such an individual has almost zero chance of death. In order to get a more sensible survival plot we must specify values of the covariates. The vector `typical` specifies an individual without edema `edtrt=0`, with albumin level of 3.5 `(log alb)=(log 3.5)`, prothrombin time 10s `(log protime)=(log 10)`, bilirubin 1.75 `(log bili)=(log 1.75)` and age 50 years. This gives a predicted survival curve with median around 10 years.

We can see the influence of edema on this curve by looking at survival curves for the model stratified on `edtrt`. The vector `typical2` just omits the `edtrt` component of `typical`. For better comparison we plot all three curves on the same graph.

We may be more interested in survival at various numbers of years, rather than the 1000 day intervals that XLISP-Stat provides on the x-axis. To do this, we obtain the survival curve for the `typical` individual as a function closure and evaluate it at 1, 2, ..., 10 years.

The full effect of covariates on the survival function can be seen dynamically using `:slider-plot`, which creates a automatically updated plot with sliders to control the covariate values. We can see how the predicted survival varies from less than one year (patients who now would be given liver transplants) to better than a decade. This level of prediction is very unusual in biomedical survival data.

17 ⟨*eg:model summaries* 17⟩≡

```
(wait)
(print-matrix (send mayo-model :conf-interval))
(wait)
(send mayo-model :plot-survival)
(wait)
(def typical (list 0 (log 3.5) (log 10) (log 1.75 ) 50))
(send mayo-model :plot-survival :xvec typical)
(wait)
(def typical2 (list (log 3.5) (log 10) (log 1.75 ) 50))
(send mayo-model-stratified :plot-survival :xvec typical2 :one-plot t)
(wait)

(def sfun (send mayo-model :survival-function-closure :xvec typical))
(funcall sfun (* 365.25 (iseq 1 10)))
(wait)

(send mayo-model :slider-plot)
```

2.4 Expected survival for groups

The survival-function methods in the previous section give the expected survival for an individual with given covariates. It is also of interest to examine the expected survival for a group of patients with differing covariates. The expected survival curve for the group will be initially steeper than that for an individual with the mean covariate vector, and will then flatten out as the higher-risk individuals are removed.

The 'exact' or 'direct-adjusted' expected survival curve is obtained by averaging the predicted survival curves for each individual. This gives the correct predicted survival curve for the group, and so is appropriate when making predictions. When censoring depends on the covariates, the direct-adjusted survival curve is consistent and the Kaplan–Meier estimator is not. There is a more complicated procedure, called Hakulinen's method (Hakulinen, 1982) that produces an estimator with the same bias as the Kaplan–Meier, but this has not been implemented here.

The methods `:group-survival-function`, and `:plot-group-survival-function` perform these calculations. They take as arguments covariates `:new-x` specified as a matrix, list of vectors or model formula object, strata `:new-strata`. As with the individual survival curves there is an optional argument `:product-limit` that chooses between the two asymptotically equivalent estimators for the baseline survival function.

The expected survival curves are also useful for model validation, at least when censoring is not strongly dependent on the covariates, as described in the next section.

18 ⟨*expected survival* 18⟩≡

```
(defmeth cox-regression-proto :plot-group-survival-function
  (&key new-x new-strata (latent-times nil) (product-limit t))
"Method arguments: &key new-x (latent-times nil) (product-limit t)
Plots expected survival for cohort with covariates given by :new-x, using
the direct-adjusted method if :latent-times is nil and Hakulinen's method
otherwise."
(let* ((curve (send self :group-survival-function :new-x new-x
                    :new-strata new-strata :latent-times latent-times
                    :product-limit product-limit))
       (plot (plot-steps (first curve)
                         (second curve)
                         :title "Group Expected Survival"
                         :variable-labels (list "time" "proportion surviving")
                         ))
       )
  (send plot :range 1 0 1)
  plot
)
)



(defmeth cox-regression-proto :group-survival-function
  (&key new-x new-strata (latent-times nil) (product-limit t))
"Method arguments: &key new-x (product-limit t)
Returns expected survival for cohort with covariates given by :new-x, using
the direct-adjusted method "
```

```
  (let* ((x (cond ((matrixp  new-x) (row-list  new-x))
                  ((objectp  new-x) (send new-x :design-matrix))
                  ((and (sequencep  new-x) (numberp (elt  new-x 0)))
                   (row-list (bind-columns  new-x)))
                  ((and (sequencep  new-x) (sequencep (elt  new-x 0)))
                   (row-list (apply #'bind-columns  new-x)))
             )
         ))
    (send self :exact-group-surv x new-strata product-limit))
  )



(defmeth cox-regression-proto :exact-group-surv (x strata product-limit)
  (let* ((beta (send self :beta))
         (stratlist (unique strata))
         (alltimes (sort-data (unique (send self :times))))
         (survs  (if strata
                     (mapcar #'(lambda (stratum)
                                 (send self :survival-function-closure
                                       :stratum stratum
                                       :product-limit product-limit)
                                 )
                             stratlist
                             )
                   (list (send self :survival-function-closure
                               :product-limit product-limit))
                   ))
         (rr (mapcar #'(lambda (xi) (exp (inner-product xi beta))) x))
         (rrstrat (if strata
                      (map-elements #'(lambda (stratum)
                                        (select rr (argselect strata stratum)))
                                    stratlist)
                    (list rr)))
         (ff (lambda (rrs sts)
               (map-elements #'(lambda (sti) (sum (^ sti rrs))) sts)))
         (basesurvs (mapcar
                     #'(lambda (fstrat) (funcall fstrat alltimes))
                     survs))
         (survcurvs (mapcar
                     #'(lambda (rrs sts) (funcall ff rrs sts))
                     rrstrat basesurvs))
         )
    (list alltimes  (/ (apply #'+ survcurvs) (length x)))
    )
  )
```

2.4.1 Examples

The Mayo model has been extensively validated by checking its predictions against actualsurvival for a number of other cohorts of patients. One such cohort is the non-randomised patients at the Mayo Clinic over the same time period, contained in the file `pbc-new.lsp`.

Here we divide these patients into three approximately equal groups based on their predicted risk under the Mayo model and plot the Kaplan-Meier and exact predicted survival for each group. A plot of this type can be found on page 195 of Fleming & Harrington (1991).

20 ⟨*eg:expected survival examples* 20⟩≡

```
(wait)
(load "pbc-new.lsp")
(def risks (matmult
            (bind-columns edtrt1 (log alb1) (log protime1)
                          (log bili1) age1)
            (send mayo-model :beta)))

(def low (which (> 5.5 risks)))
(def high (which (< 6.5 risks)))
(def med (which
          (map-elements #'(lambda (x y)
                            (and x y))
                        (> 6.5 risks) (< 5.5 risks)
                        )
          )
     )
(def lowrisk (send mayo-model :group-survival-function
                   :new-x (mapcar #'(lambda (l)
                                      (select l low))
                                  (list edtrt1 (log alb1)
                                        (log protime1) (log bili1)
                                        age1
                                        )
                                  )
                   )
     )
(def medrisk (send mayo-model :group-survival-function
                   :new-x (mapcar #'(lambda (l)
                                      (select l med))
                                  (list edtrt1 (log alb1)
                                        (log protime1) (log bili1)
                                        age1
                                        )
                                  )
                   )
     )
(def hirisk (send mayo-model :group-survival-function
```

```
                  :new-x (mapcar #'(lambda (l)
                                     (select l high))
                                 (list edtrt1 (log alb1)
                                       (log protime1) (log bili1)
                                       age1
                                       )
                                 )
                  )
     )

(def gtime1 (mapcar #'(lambda (i)
                        (select time1 i))
                    (list low med high)
                    )
     )
(def gstat1 (mapcar #'(lambda (i)
                        (select status1 i))
                    (list low med high)
                    )
     )
(def kmplots (mapcar #'(lambda (tt ss)
                         (km-plot tt ss (kmest tt ss)))
                     gtime1
                     gstat1
                     )
     )

(def validation
     (plot-lines (elt kmplots 0)
                 :type 'D
                 :title "Predictive accuracy of Mayo Model"
                 :variable-labels (list "Days on study"
                                        "Proportion surviving")
                )
     )
(send validation :add-lines (elt kmplots 1) :type 'D)
(send validation :add-lines (elt kmplots 2) :type 'D)
(send validation :range 1 0 1)
(send validation :add-lines lowrisk :color (elt *colors* 2))
(send validation :add-lines medrisk :color (elt *colors* 2))
(send validation :add-lines hirisk :color (elt *colors* 2))
```

2.5 Diagnostics

Diagnostic methods for the Cox model are based on the martingale residuals

$$M_i(\infty) = \int_0^\infty dM_i(t)$$

and the score process increments

$$dU_i(t) = (z_i(t) - \bar{z}(t))dM_i(t),$$

the contribution of individual i at time t to the derivative of the partial loglikelihood, where $\bar{z} = S^{(1)}(t, \hat{\beta})/S^{(0)}(t, \hat{\beta})$ is the risk-weighted average covariate vector. This can be summed over individuals to yield the Schoenfeld residuals $(z_i(t) - \bar{z}_i(t))d\bar{N}(t)$ or integrated over time to yield score residuals

$$\int_0^\infty z_i(t) - \bar{z}_i(t)dM_i(t).$$

Smoothed plots of martingale residuals against variables not in the model give estimates of the correct functional form of these variables. This can be done with the `:plot-new-variable` methods.

The score residuals, standardised by multiplying by the inverse of the information matrix, give approximate jackknife residuals ($\Delta\beta$s) for each observation. These are returned by the `:delta-betas` method and can be plotted with `:plot-delta-betas`. When there is more than one record with the same `id` it is possible to compute $\Delta\beta$s deleting each record or all records for that observation. The former may be appropriate for clustered or multivariate survival data; the latter is more suitable for recurrent events or when the multiple records merely reflect time-dependence in the covariates.

Schoenfeld residuals, scaled by the inverse of the information matrix and smoothed against time, give an approximation to the difference between the overall estimated $\hat{\beta}$ and an estimate based just on a window of data around each time (Grambsch & Therneau, 1994). We add $\hat{\beta}$ to these to give the `:scaled-schoenfeld-residuals` which can be plotted and smoothed to give an estimate of a time-varying coefficient $\hat{\beta}(t)$ with `:plot-ph-test` or used to construct a formal test of the proportional hazards assumption with `:ph-test`. There is a Schoenfeld residual for each variable at each failure time and they are returned as a list of sequences, with the first sequence being the times.

The formal test based on the scaled Schoenfeld residuals (Grambsch & Therneau, 1994) is based on the correlation of the residuals with some function of time where the function is specified by the `:scaled` argument to `:ph-test`. The default here is the identity transformation. A similar class of tests may be based on unscaled Schoenfeld residuals, but these are less desirable as time-dependence in one variable may "leak across" into another strongly correlated variable. The function of time for these tests may be specified with the `:unscaled` argument. These tests are provided for compatibility, as the `:unscaled #'rank` test proposed by Harrell & Lee (1986) has been implemented in a SAS macro and in the SPIDA package. It is available as `:harrell-lee-test`. The proportional hazards tests return a matrix with columns for the correlation, t-statistic and p-value.

When a new variable is to be added to the model the appropriate functional form can be assessed by fitting a model without that variable and smoothing the martingale residuals against the variable in question (Fleming & Harrington, 1991). These is done by `:plot-new-variable`.

Expected survival curves can be used to compare the model predictions with the Kaplan–Meier estimate for a subset of the original data or for new and independent data. The `:scatterplot-expected` and `:scatterplot-predicted` methods do this. The first is for subsets of the original data. It takes one argument, `:covariates`, which is used to construct a scatterplot matrix linked to a graph of the Kaplan–Meier and expected survival estimates. Selecting or brushing subsets of the data in the scatterplot matrix causes the survival graphs to be recomputed.

This is useful for examining the model fit to subsets based on covariates used to fit the model, as a check on functional form, or based on new covariates, as a check on their importance. Brushing points is computationally demanding, with the smoothness of the mouse response being marginal on a 150Mhz Pentium Pro and good on a 300Mhz Pentium II computer. Selecting subsets is feasible on much slower computers.

The second method, `:scatterplot-predicted`, performs the same function for new data. Its arguments are `:covariates`, `:new-x :new-times` and `:new-status`; where `:covariates` are covariates for the scatterplot matrix and `:new-x` are the values of the covariates used to fit the model. If `:covariates` is omitted, `:new-x` will be plotted instead. These two graphical diagnostics are related to methods described by Atkinson (1995) and some code written to accompany that paper, obtained from the UCLA XLISP-Stat archive, is used in the file `expsurv1.lsp`.

23 ⟨*diagnostics* 23⟩≡

```
(defmeth cox-regression-proto :martingales (&optional (new nil set))
"Message ars: (&optional new)
Sets or returns martingale residuals"
  (when set
        (setf (slot-value 'martingales) new)
        )
  (slot-value 'martingales))


(defmeth cox-regression-proto :scaled-schoenfeld-residuals ()
  (let* ((ri (send self :schoenfeld-residuals))
         (n (length ri))
         (betas (matmult (bind-columns (repeat 1 n))
                         (bind-rows (send self :beta))))
         (scaled  (+ betas
                     (matmult  (apply #'bind-rows (mapcar #'second ri))
                               (* n (send self :covariance-matrix)))))
         )
    (cons  (mapcar #'first ri) (column-list scaled))
    )
  )


(defmeth cox-regression-proto :plot-ph-test (&optional vars)
"Arguments: vars
Plots smoothed scaled Schoenfeld residuals"
(let* (
       (r (send self :scaled-schoenfeld-residuals))
       (vns (send self :predictor-names))
       (varlist (or vars (iseq (length (send self :beta)))))
       )
  (map-elements
   #'(lambda (v)
```

```
        (scatter-smooth (first r) (elt r (+ 1 v))
                        :title
                        (format nil "Proportional hazards: ~a" (elt vns v))
                        :variable-labels
                        (list "Time" "Scaled Schoenfeld Residuals"))) varlist)
  )
)

(defmeth cox-regression-proto :harrell-lee-test ()
"Args:
Harrell & Lee's test for proportional hazards
based on the Schoenfeld residuals"
  (second (send self :ph-test :scaled nil :unscaled #'rank))
)

(defmeth cox-regression-proto :ph-test (&key (scaled #'identity) (unscaled nil))
"Args: &key (scaled #'identity) (unscaled nil)
Tests for proportional hazards based on scaled or raw Schoenfeld residuals"
(list (cond ((null scaled) nil)
            ((functionp scaled)
             (let* ((sr (send self :scaled-schoenfeld-residuals))
                    (srs (cons (funcall scaled (first sr)) (cdr sr)))
                    (cmat (apply #'covariance-matrix srs))
                    (sds (/ (sqrt (diagonal cmat))))
                    (rmat (matmult (diagonal sds) cmat (diagonal sds)))
                    (rhos (cdr (coerce (first (row-list rmat)) 'list)))
                    (n2 (- (length (first srs)) 2))
                    (ts (* rhos (sqrt n2)))
                    (names (send self :predictor-names))
                    (ps (* 2 (t-cdf (- (abs ts)) n2)))
                    )
               (bind-columns names rhos ts ps)
               )
             )
            )
      (cond ((null unscaled) nil)
            ((functionp unscaled)
             (let* ((sr (send self :schoenfeld-residuals))
                    (srs (cons (funcall unscaled (firsts sr))
                               (column-list (apply #'bind-rows (seconds sr)))))
                    (cmat (apply #'covariance-matrix srs))
                    (sds (/ (sqrt (diagonal cmat))))
                    (rmat (matmult (diagonal sds) cmat (diagonal sds)))
                    (rhos (cdr (coerce (first (row-list rmat)) 'list)))
                    (n2 (- (length (first srs)) 2))
                    (ts (* rhos (sqrt n2)))
```

```
                    (names (send self :predictor-names))
                    (ps (* 2 (t-cdf (- (abs ts)) n2)))
                    )
               (bind-columns names rhos ts ps)
               )
             )
            )
      )
)
(defmeth cox-regression-proto :delta-betas (&key (total t))
"Args: &key (total t)
Returns a list of sequences containing delta-betas.  If total is t the delta-betas are
collapsed over records with the same id and the first sequence is the unique ids; if
total is nil the ordering is the same as in the data"
(let* ((id (send self :id))
       (unique-id (unique id))
       (scores (or (send self :scores)
                   (error "Can't calculate delta-betas - no martingale residuals")))
       (invinfomat (send self :model-covariance-matrix))
       (partial-db (row-list (matmult scores invinfomat )))
       (db  (cond
             (total
              (mapcar #'(lambda (i)
                          (apply #'+ (select partial-db (argselect id i))))
                      unique-id))
             (t
              partial-db)))
       )
  (if total
      (cons unique-id (column-list (apply #'bind-rows db)))
    (column-list (apply #'bind-rows db))
    )
  )
)

(defmeth cox-regression-proto :plot-delta-betas (&key (total t) (link t) (vars nil))
"Args: &key (total t) (link t) (vars nil)
Plots delta-betas in linked plots labelled by id. A subset of the variables
can be chosen with vars.
The default is to sum the delta-betas over all records with
the same id, set total to nil to get separate delta-betas for different records."

(let* ((db (send self :delta-betas :total total))
       (idlabel (map-elements #'write-to-string
                              (if total
                                  (first db)
```

```
                                (send self :id))))
        (titles (send self :predictor-names))
        (ylabels (mapcar #'(lambda (title)
                             (format nil "dBeta for ~a" title)) titles))
        (rval (mapcar #'(lambda (db title ylabel)
                          (index-plot db
                                      :title title
                                      :point-labels idlabel
                                      :variable-labels (list "index" ylabel)))
                      (if total
                          (cdr db)
                        db)
                      titles
                      ylabels))
        )
  (when link (apply #'link-views rval))
  rval
))


(defmeth cox-regression-proto :plot-new-variable (newx &optional label)
"Args: newx &optional label
 Smooth plot of variable newx against martingale residuals, where newx should be a
 variable not in the model.  The smooth estimates the correct functional form of newx"
(let* ((rmart (send self :martingales))
       (xlabel (or label "New covariate"))
       )
  (scatter-smooth newx
                  rmart
                  :variable-labels (list xlabel "Martingale residuals")
                  :point-labels (mapcar #'write-to-string (send self :id)))
  )
)


(require "expsurv1")
(defmeth  cox-regression-proto :scatterplot-expected
  (&key covariates covariate-labels)
"Args: &key covariates covariate-labels
Scatterplot matrix of covariates linked to expected & K-M survival graph"
  (let* ((scat (scatterplot-matrix covariates
                                   :variable-labels covariate-labels))
         (time (send self :times))
         (status (send self :status))
         (coxobject self)
         (data (km-plot time status (kmest time status)))
```

```
          (km (plot-lines (first data)
                          (second data)
                          :title "Survival Plot"))
          (current-points (iseq (length time))))
      (send km :range 1 0 1)
      (send scat :point-state (iseq (length time)) 'hilited)
      (defmeth scat :unselect-all-points ()
        (setf current-points nil)
        (send km :clear)
        (call-next-method))
      (defmeth scat :adjust-points-in-rect (x1 y1 w h s)
        (let* ((p-i-r (send self :points-in-rect x1 y1 w h)))
          (setf current-points
                (case (send self :mouse-mode)
                  (brushing p-i-r)
                  (selecting (union p-i-r current-points))))
          (if current-points
              (prog*
               ((points (sort-data current-points))
                (time-sel (select time points))
                (status-sel (select status points))
                (expected (send coxobject
                                :group-survival-function
                                :new-x  (mapcar
                                         #'(lambda (x)
                                             (select x points))
                                         (column-list (send coxobject :x)))
                                :new-strata
                                (select (send coxobject :strata) points)))
                (new-plot (km-plot time-sel status-sel
                                   (kmest time-sel status-sel))))
               (send km :clear :draw nil)
               (send km :add-lines (first expected) (second expected))
               (send km :add-lines
                     (first new-plot) (second new-plot))))
          (call-next-method x1 y1 w h s)))
      scat))

(defmeth  cox-regression-proto :scatterplot-predicted
  (&key (covariates nil) covariate-labels  x times status (strata nil))
"Args: &key (covariates nil)  x times status (strata nil)
Scatterplot matrix of covariates linked to predicted & K-M survival graph"
  (let* ((scat (scatterplot-matrix (or covariates x)
                                   :variable-labels covariate-labels))
         (time times)
         (coxobject self)
```

```
      (strata (or strata (* times 0)))
      (data (km-plot time status (kmest time status)))
      (km (plot-lines (first data)
                      (second data)
                      :title "Survival Plot"))
      (current-points (iseq (length time))))
(send km :range 1 0 1)
(send scat :point-state (iseq (length time)) 'hilited)
(defmeth scat :unselect-all-points ()
  (setf current-points nil)
  (send km :clear)
  (call-next-method))
(defmeth scat :adjust-points-in-rect (x1 y1 w h s)
  (let* ((p-i-r (send self :points-in-rect x1 y1 w h)))
    (setf current-points
          (case (send self :mouse-mode)
            (brushing p-i-r)
            (selecting (union p-i-r current-points))))
    (if current-points
        (prog*
         ((points (sort-data current-points))
          (time-sel (select time points))
          (status-sel (select status points))
          (expected (send coxobject :group-survival-function
                          :new-x   (mapcar
                                    #'(lambda (xi)
                                        (select xi points))
                                    x)
                          :new-strata (select strata points)))
          (new-plot (km-plot time-sel status-sel
                             (kmest time-sel status-sel))))
         (send km :clear :draw nil)
         (send km :add-lines (first expected)
               (second expected))
         (send km :add-lines
               (first new-plot) (second new-plot))))
    (call-next-method x1 y1 w h s)))
scat))
```

2.5.1 Examples

These examples again relate to the Mayo model for primary biliary cirrhosis. First we test the proportional hazards assumption with `:ph-test`. We use the log transformation of time. For comparison we also use the Harrell & Lee test.

The form of departure from proportional hazards can be assessed with `:plot-ph-test`. The edema `edtrt` variable has a very sharply decreasing effect over the first 2 years or so or so, and little to no predictive power after that time. The prothrombin time variable `(log protime)` has a wildly non-constant effect. Note that the smoothed curve estimates a time-varying $\hat{\beta}(t)$, the logarithm of the hazard ratio, so the variation in the curve for `(log protime)` from below zero to near 10 is very extreme.

The Mayo model contains a log-transformation of the bilirubin variable. This can be chosen by fitting a model without `bili` and then using `:plot-new-variable`. Smoothing residuals against `bili` suggests a log-transformation, and this is confirmed by the much more linear plot when smoothing against `(log bili)`.

Influential points can be detected with `:plot-delta-betas`. Use the plot menu to turn on linking and variable label. There are particularly extreme values in `edtrt` (id 292), `(log protime)` (id 106), `(log bili)` (id 80), and `age` (id 252). Two of these four anomalous values are in fact data entry errors. Rechecking these five variables for all 312 cases did not turn up any other errors (Fleming & Harrington (1991)).

Selecting the different stages in the scatterplot-matrix expected survival plot confirms that `stage` has little additional predictive power, which is good, since histologic stage can only be measured by liver biopsy.

As described in section 2.4.1 the Mayo model was extensively vaidated against independent data from other cohorts of patients, including the non-randomised patients from the Mayo cohort. The `:scatterplot-predicted` method gives a dynamic version of these comparisons allowing you to explore subsets of the new data and evaluate the model fit. For example, overfitting of the original model would tend to cause overestimation of the survival for good risk patients and underestimation for poor risk patients. Selecting high and low subsets based on serum bilirubin or prothrombin time would reveal this. In fact the model appears to fit very well.

29 ⟨*eg:diagnostic examples* 29⟩≡

```
(wait)
(def mayo-ph (send mayo-model :ph-test :scaled #'log))
(print-matrix (first mayo-ph))

(print-matrix (send mayo-model :harrell-lee-test))
(wait)

(send mayo-model :plot-ph-test)
(wait)

(def mayo-formula-no-bili (as-formula '((term age) (term (log protime))
                                        (term (log alb)) (term edtrt))))

(def mayo-no-bili (cox-model  :times time :status status
                              :x mayo-formula-no-bili))

(send mayo-no-bili :plot-new-variable bili "Bilirubin")
(wait)

(send mayo-no-bili :plot-new-variable (log bili) "log(Bilirubin)")
```

```
(wait)

(send mayo-model :plot-delta-betas)
(wait)

(send mayo-model :scatterplot-expected :covariates (list stage copper sgot)
                :covariate-labels (list "Stage" "urine copper" "SGOT"))
(wait)

(load "pbc-new")
(send mayo-model :scatterplot-predicted
      :x (list edtrt1 (log alb1) (log protime1) (log bili1) age1)
      :times time1
      :status status1
      :covariate-labels (list "edema" "log alb" "log protime" "log bili" "age")
      )
```

2.6 Tie handling

It is reasonably common for more than one subject to have the same observed failure time, usually because the times are recorded to the nearest year, month or day. The exact Cox partial likelihood is computationally difficult when more than one failure occurs at a given time and is usually not the default choice in implementations of the method. There are two popular approximations that are easier to compute. Most software uses by default an approximation attributed variously to Breslow and Peto that would be appropriate for genuinely simultaneous events. Another approximation due to Efron is used by default in the `survival` packages for S writted by Terry Therneau, who claims it is considerably more accurate.

This package implements both of these approximations as objects of class `cox-tie-proto`. Other approximations, or the exact partial likelihood, could be similarly implemented. The prototype `cox-tie-proto` implements the Peto/Breslow approximation, and `breslow-ties`, `peto-ties` and `counting-process-ties` are aliases for this. The Efron approximation is performed by `efron-ties`. The default tie-handling object is stored in `*cox-default-ties*`, and is initially `efron-ties`.

There is no difficulty with tied entry times or tied censoring times as these do not contribute to the partial likelihood. Some convention is needed when entry, censoring and failure times are all simultaneous. It is standard in this case to assume that the failures precede the censorings, and this fits in well with the counting process formulation of survival analysis. There is no standard convention on entry times. Assuming that entry times precede failures at the same instant allows an individual to enter and leave the data set on the same day, when data are recorded to the nearest day, which is convenient. On the other hand, consistency with the counting process formulation would suggest that entry times follow failures. In this package it is assumed that entries precede failures at the same recorded time. Any problem with this can be solved by adding 0.5 to all the entry times.

Tie handling objects respond to five messages: `:increment`, `:dLambda`, `:schoenfeld`, `:dloglike` and `:prettyprint`.

31 ⟨*tie-handling objects* 31⟩≡

```
(defproto cox-tie-proto)
(defproto breslow-ties () () cox-tie-proto)
(defproto counting-process-ties () () cox-tie-proto)
(defproto peto-ties () () cox-tie-proto)
(defproto efron-ties () () cox-tie-proto)
```

⟨*ties: increment weighted moments* 33⟩
⟨*ties: increment of baseline hazard* 35a⟩
⟨*ties: increment of loglikelihood* 32⟩
⟨*ties: schoenfeld residuals* 34⟩
⟨*ties: pretty-print name* 35b⟩

The two approximate likelihoods both involve the product of the risk $\exp(\beta' z)$ for the tied failures divided by an average of $\exp(\beta' z)$ over those at risk. For the Efron approximation the failing individuals are considered only partly at risk for each event; for the Peto/Breslow approximation they are considered fully at risk.

32 ⟨*ties: increment of loglikelihood* 32⟩≡

```
(defmeth cox-tie-proto :dloglike (zlist betahat ns0)
"Args: zlist betahat ns0
Computes the increment in the loglikelihood"
(let* ((btz (map-elements #'(lambda (z) (inner-product z betahat)) zlist))
       )
  (- (apply #'+ btz) (* (length zlist) (log ns0)))
  )
)
(defmeth efron-ties :dloglike (zlist betahat ns0)
"Args: zlist betahat ns0
Computes the increment in the loglikelihood"
(let*  (
        (btz (map-elements #'(lambda (z) (inner-product z betahat)) zlist))
        (expbtz (exp btz))
        (n (length zlist))
        (w (/ (iseq  n) n))
        (ebtzsum (apply #'+ expbtz))
        (btzsum (apply #'+ btz))
        )
  (- btzsum (sum (log  (- ns0 (* w ebtzsum)))))
  )
)
```

The `:increment` message returns the increments of the score vector and information matrix at the given time.

33 ⟨*ties: increment weighted moments* 33⟩≡

```
(defmeth cox-tie-proto  :increment (zlist betahat ns0 ns1 ns2)
(let* (
       (expbtz (map-elements #'(lambda (z) (exp (inner-product z betahat))) zlist))
       (zbtz (* expbtz zlist))
       )
   (list (- (apply #'+ zlist) (* (length zlist) (/ ns1 ns0)))
         (* (length zlist) (- (/ ns2 ns0) (outer-product (/ ns1 ns0) (/ ns1 ns0)))))
  )
)
(defmeth efron-ties :increment (zlist betahat ns0 ns1 ns2)
(let* (
       (expbtz (map-elements #'(lambda (z) (exp (inner-product z betahat))) zlist))
       (zbtz (* expbtz zlist))
       (z2btz (* expbtz (map-elements #'outer-product zlist zlist)))
       (n (length zlist))
       (w (/ (iseq  n) n))
       (zsum (apply #'+ zbtz))
       (z2sum (apply #'+ z2btz))
       (ebtzsum (apply #'+ expbtz))
       (zbars (/ (mapcar #'(lambda (wi) (- ns1 (* wi zsum))) w)
                 (- ns0 (* w ebtzsum))))
       (z2bars (/ (mapcar #'(lambda (wi) (- ns2 (* wi  z2sum))) w)
                  (- ns0 (* w ebtzsum))))
       )
   (list (- (apply #'+ zlist) (apply #'+ zbars))
         (- (apply #'+ z2bars) (apply #'+ (mapcar #'outer-product zbars zbars))))
   )
)
```

The Schoenfeld residual measures the difference between the covariate vector of the observed failure and the weighted average of those at risk. This weighted average is different for the different tie approximations.

34 ⟨*ties: schoenfeld residuals* 34⟩≡

```
(defmeth cox-tie-proto :schoenfeld (zlist betahat ns0 ns1)
"Args: zlist betahat
Computes the appropriate Schoenfeld residual(s) for the tied failures"
(let* ((expbtz (map-elements #'(lambda (z) (exp (inner-product z betahat))) zlist))
       (n (length zlist))
       )
   (- zlist (repeat (list (/ ns1 ns0)) n))
  )
)
(defmeth efron-ties :schoenfeld (zlist betahat ns0 ns1)
"Args: zlist betahat
Computes the appropriate Schoenfeld residual(s) for the tied failures"
(let*  (
        (expbtz (map-elements #'(lambda (z) (exp (inner-product z betahat)))
                              zlist))
        (zbtz (* expbtz zlist))
        (n (length zlist))
        (w (/ (iseq  n) n))
        (zsum (apply #'+ zbtz))
        (ebtzsum (apply #'+ expbtz))
        (zbars (/ (mapcar #'(lambda (wi) (- ns1 (* wi zsum))) w)
                  (- ns0 (* w ebtzsum))))
        )
     (- zlist zbars)
  )
)
```

The increment of the baseline hazard at a failure time is the reciprocal of an average over the risk set, and so depends on the approximation used.

35a ⟨*ties: increment of baseline hazard* 35a⟩≡

```
(defmeth cox-tie-proto :dLambda (zlist betahat ns0)
"Args: zlist betahat ns0
Computes the increment of the baseline cumulative hazard for the tied failures.
equiv-at-risk is the sum of exp(b'z) just before this time, so
(/ nS0) is the increment if there is only one failure"
  (/ (length zlist) ns0)
)
(defmeth efron-ties :dLambda (zlist betahat ns0)
"Args: zlist betahat ns0
Computes the increment of the baseline cumulative hazard for the tied failures.
equiv-at-risk is the sum of exp(b'z) just before this time, so
(/ nS0) is the increment if there is only one failure"
(let*  (
        (expbtz (map-elements #'(lambda (z) (exp (inner-product z betahat)))
                              zlist))
        (n (length zlist))
        (w (/ (iseq  n) n))
        (ebtzsum (apply #'+ expbtz))
        )
  (sum (/ (- ns0 (* w ebtzsum))))
  )
)
```

The `:prettyprint` message is used by the `:display` methods of `:cox-regression-proto` to tell the user what tie approximation has been used.

35b ⟨*ties: pretty-print name* 35b⟩≡

```
(defmeth cox-tie-proto :prettyprint (&optional (stream t))
  (format stream "Tie-handling prototype~%")
)
(defmeth peto-ties :prettyprint (&optional (stream t))
  (format stream "Peto's (counting process) approximation~%")
)
(defmeth breslow-ties :prettyprint (&optional (stream t))
  (format stream "Breslow's (counting process) approximation~%")
)
(defmeth counting-process-ties :prettyprint (&optional (stream t))
  (format stream "Counting process (Breslow's approximation)~%")
)
(defmeth efron-ties :prettyprint (&optional (stream t))
  (format stream "Efron's approximation~%")
)
```

2.6.1 Examples

The choice of tie handling typically makes very little difference, as shown here for the Mayo PBC data. We fit the Mayo model with both tie handling methods and then break ties randomly to get an untied data set, taking care not to make any other changes to the ordering of events and censorings.

36 ⟨*eg:tie examples* 36⟩≡

```
(wait)
(cox-model :times time :status status :x mayo-model-formula :ties efron-ties)
(cox-model :times time :status status :x mayo-model-formula :ties breslow-ties)
(def tt (- time (* 0.01 status (uniform-rand 312))))
(cox-model :times tt :status status :x mayo-model-formula)
```

2.7 One and two-sample functions

The Kaplan–Meier survival function estimate and the Nelson–Aalen cumulative hazard estimate are obtained by fitting a stratified Cox model without covariate effects. This allows code reuse.

The `(cox-score-test)` function is an approximation to the logrank test. In the absence of tied failures it gives the logrank test, in the presence of ties it gives Peto's conservative approximate logrank test. For an ordered variable the `:trend t` argument requests a score test for linear trend, which is more powerful against ordered alternatives.

37 ⟨*one and two sample functions* 37⟩≡

```
(defun kaplan-meier
(&key times status (entry nil) (groups (repeat 0 (length times))))
"Args: &key times status groups entry"
(let* ((fakex (uniform-rand (length times)))
       (nullcox (cox-model :x fakex :times times :entry entry :status status
                           :strata groups :verbose nil :print nil :print-summary nil
                           :count-limit 1 :suppress-warnings t))
       )
   (send nullcox :compute)
   (send nullcox :survival-function)
  )
)

(defun plot-kaplan-meier
  (&key times status (entry nil) (groups  (repeat 0 (length times))) inverted (one-plot t))
"Args: &key times status groups entry inverted (one-plot t)"
(let* ((fakex (uniform-rand (length times)))
       (nullcox (cox-model :x fakex :times times :entry entry :status status
                           :strata groups :verbose nil :print nil :print-summary nil
                           :count-limit 1 :suppress-warnings t))
       )
   (send nullcox :compute)
   (send nullcox :plot-survival :inverted inverted :one-plot one-plot)
   )
)

(defun nelson-aalen
  (&key times status (entry nil)(groups  (repeat 0 (length times))))
"Args: &key times status groups entry "
(let* ((fakex (uniform-rand (length times)))
       (nullcox (cox-model :x fakex :times times :entry entry :status status
                           :strata groups :verbose nil :print nil :print-summary nil
                           :count-limit 1 :suppress-warnings t))
       )
  (send nullcox :compute)
  (send nullcox :cumulative-hazard)
  )
```

```
)

(defun plot-nelson-aalen
  (&key times status (entry nil) (groups  (repeat 0 (length times))) (one-plot t))
"Args: &key times status groups entry (one-plot t)"
(let* ((fakex (uniform-rand (length times)))
       (nullcox (cox-model :x fakex :times times :entry entry :status status
                           :strata groups :verbose nil :print nil :print-summary nil
                           :count-limit 1 :suppress-warnings t))
       )
  (send nullcox :compute)
  (send nullcox :plot-cumulative-hazard :one-plot one-plot)
  )
)

(defun cox-score-test
  (&key groups times status (entry nil) (trend nil) (strata (repeat 0 (length times))))
"Args: &key groups times status entry (trend nil)
Peto's approximate Logrank test for differences between groups.
Requires modelformula extensions."
(let* (
       (gmodel (factor groups :namebase "Group "))
       (gmat (first gmodel))
       (gnames (second gmodel))
       (p (second (array-dimensions gmat)))
       (x (if trend (matmult gmat (iseq 1 p)) gmat))
       (nullcox (cox-model :x x :times times :entry entry :status status :strata strata
                           :verbose nil :print nil :print-summary nil :count-limit 1
                           :suppress-warnings t :ties peto-ties))
       )
  (send nullcox :compute)
  (let* (
         (beta (send nullcox :beta))
         (infomat (inverse (send nullcox :covariance-matrix)))
         (score (matmult beta infomat))
         (grouptests (/ score (sqrt (diagonal infomat))))
         (overalltest (inner-product score beta))
         )
    (format t (if trend "Logrank Trend Test~%" "Logrank test~%"))
    (when (and (> p 1) (null trend))
          (dotimes (i p)
                   (format t "Group ~a.  Z=~6,4g   p=~6,4g~%"
                           (elt gnames i)
                           (elt grouptests i)
                           (* 2 (- 1 (normal-cdf (abs (elt grouptests i))))))
                   )
```

```
          )
      (format t "Overall test: X^2=~8,6g   p=~6,4g~%"
              overalltest
              (- 1 (chisq-cdf overalltest p)))
      (if (< (length (unique times)) (length times))
          (format t "Note: Tied times present -- results slightly conservative~%"))
      (list grouptests overalltest)
      )
    )
  )
```

2.7.1 Examples

Again we use the Mayo Clinic PBC data, and this time examine the effectiveness of the randomised tratment, D-penicillamine by plotting survival curves and performing an approximate logrank test.

39 ⟨*eg:basic examples* 39⟩≡

```
(wait)
(cox-score-test :groups trt :times time :status status)
(wait)
(plot-kaplan-meier :groups trt :times time :status status)
```

2.8 Time-dependent covariates

Suppose that, to investigate time-dependence, we want to fit a different covariate effect before and after three years in the Mayo PBC data. Individuals who survive more than three years now have two records, one beginning at 0 days and censored at 1096 days and one beginning at 1096 days and going until the observation time. In order to estimate separate covariate effects for a given variable, such as `(log protime)`, we construct a variable that is 0 for the first record and equal to `(log protime)` for the second, or *vice versa*.

With any time-dependence the survival and hazard function estimates are less useful, and in particular the functions that give hazard or survival estimates for particular covariate vectors may not be meaningful, as the survival curve for a constant covariate vector may not correspond to any possible individual.

More generally we can specify any form of time-dependence using lisp functions. The second example allows the effect of `(log protime)` to be a linear function of time, and modifies the age variable to give current age rather than age at time 0.

The implementation of arbitrary time-dependence is not as complete as that for piecewise constant covariates. The martingale residuals are not available, and as a result neither are $\Delta\beta$s or the model-agnostic covariance matrix. The model formula system cannot handle arbitrary time-dependence either.

40 ⟨*eg:time-dependence* 40⟩≡

```
(def id (iseq 312))
(def after2 (select id (which (>= time 1096))))
(def protime2 (append (repeat 1 312) (select protime after2)))
(def protime1 (append protime (repeat 1 (length after2))))
(def edtrt12 (append edtrt (select edtrt after2)))
(def alb12 (append alb (select alb after2)))
(def age12 (append age (select age after2)))
(def bili12 (append bili (select bili after2)))
(def id12 (append id after2))
(def time12 (append (mapcar #'(lambda (ti) (min ti 1096)) time)
                    (select time after2)))
(def entry12 (append (repeat 0 312) (repeat 1096 (length after2))))
(def status12 (append (mapcar #'(lambda (ti si)
                                  (if (< ti 1096) si 0)) time status)
                      (select status after2)))


(def mayo-model-3-years
     (cox-model :times time12 :status status12 :entry entry12
                :x (as-formula '((term (log protime1)) (term (log protime2))
                                 (term edtrt12)
                                 (term (log alb12)) (term age12)
                                 (term (log bili12))))
                ))

(def mayo-model-timedep
     (cox-model :times time :status status
                :x (list (log alb) (log bili) edtrt (log protime))
                :time-x (list protime age)
```

```
:time-funs (list #'(lambda (xi ti)
                     (* (elt xi 0)  (/ ti 1000)))
                 #'(lambda (xi ti)
                     (+ (elt xi 1) ti))
                 )
:predictor-names (list "log alb" "log bili" "edema"
                       "log protime" "(log protime)(t)"
                       "attained age")
))
```

2.9 Recurrent events and multivariate failure times

There is no computational difficulty in extending the Cox regression model to situations where an individual may have more than one event each. These events can be of the same type, such as recurrent infections, or of different types, such as tumour recurrence and death in an individual or disease onset for different members of a family.

This extension does raise two important statistical issues. The first is that correlations between events on the same individual make the Cox model standard errors incorrect and the second is that different choices of stratification and time scale lead to coefficients with quite different and sometimes unintuitive interpretations. Good reviews of this area are given by Therneau & Hamilton (1997), Wei & Glidden (1997) and Li & Lagakos (1997) and it is strongly recommended that you read the literature carefully before performing analyses of this sort.

The problem of correlation can be easily solved by constructing a variance matrix estimate that does not rely on the model being correct (Lin & Wei, 1989; Wei et al., 1989). This model-agnostic or model-robust variance matrix can be motivated either by a jackknife argument or by a Taylor series expansion but a rigorous proof of its validity is complicated. The variance matrix is of the information sandwich form

$$\widehat{\mathrm{var}}[\hat{\beta}] = \mathcal{I}^{-1}\left(\sum_{i=1}^{n} U_i(\hat{\beta})U_i'(\hat{\beta})\right)\mathcal{I}^{-1}$$

where $\mathcal{I}^{-1}$ is the model-based variance estimate, the inverse of the information matrix, and $U_i(\hat{\beta})$ is the score residual, the contribution of the ith individual to the derivative of the partial likelihood, evaluated at the estimate $\hat{\beta}$.

Here we show how to perform three popular types of recurrent event analysis on a data set from Appendix D of Fleming & Harrington (1991) also used by Therneau & Hamilton (1997). They come from a randomised trial of interferon-gamma, an immune system messenger, for treating chronic granulomatous disease (CGD) an complex of rare genetic disorders in which the immune response to bacterial infection is impaired. 128 patients were randomised to three injections per week of interferon-gamma or placebo, with the outcome measurement being time to first serious infection. The study stopped early when it became obvious that the treatment was successful. After all the data were collected there were 14 first serious infections in the interferon-gamma group and 30 in the control group, with a total of 20 and 56 respectively when serious infections past the first were included.

There are 203 records in the data set, containing an identification variable `id`, descriptive variables including treatment group, and the information on time to infection. For each time-to-infection there is a sequence number 1,2,3,..., a duration of time, a start and stop time and a censoring/infection indicator. The start time is zero for the first infection and for each subsequent infection is one day after the time of the previous one.

The first analysis is a simple generalisation of the Cox model in which the rate of new events for an individual who is still under observation is modelled as the product of a non-parametric baseline rate (or more strictly, *intensity*) and a function of the covariates

$$\lambda_i(t; z_i(t)) = \lambda_0(t)e^{\beta' z_i(t)}Y_i(t).$$

This is often called the Andersen–Gill model.

In the second analysis we compare individuals only to others who have had the same number of events. This is called a *conditional* or *transitional* model. A separate Cox model can be fitted to time to first event, time to second event from first event and so on, or the models can be combined and allowed to share some or all of the same coefficients. This is done by using the start and stop times in the data set and stratifying on the infection number.

In this data set there is 1 patient with 5 infections and 1 with 4 infections so it is not feasible to estimate completely different coefficients for different infection numbers. We can compromise by considering first and subsequent infections and estimating separate effects for treatment and genetic inheritance type of the disorder in the first and subsequent infections.

43 ⟨*eg:recurrent events* 43⟩≡

```
(wait)
(load "cgd")

(def cgd-formula-1 (as-formula '((factor cgd-trt))))
(def cgd-formula-2 (as-formula '((factor cgd-trt) (term cgd-age)
                                 (factor cgd-type) (factor cgd-sex)
                                 )))

(def cgd-first (pmin cgd-seqnum 2))

(def cgd-formula-1-strat (as-formula '((factor cgd-trt)
                                       (interaction (list cgd-trt cgd-first))
                                       )))

(def cgd-formula-2-strat (as-formula '((factor cgd-trt) (term cgd-age)
                                       (factor cgd-type) (factor cgd-sex)
                                       (interaction (list cgd-trt cgd-first))
                                       (interaction (list cgd-type cgd-first))
                                       )))



(wait)
(def andersen-gill-1
     (cox-model :times cgd-stop :entry cgd-start :status cgd-status
                :x cgd-formula-1
                :id cgd-id :agnostic t))

(def andersen-gill-2
     (cox-model :times cgd-stop :entry cgd-start :status cgd-status
                :x cgd-formula-2
                :id cgd-id :agnostic t))
(wait)

(def conditional-1
     (cox-model :times cgd-stop :entry cgd-start :status cgd-status
                :x cgd-formula-1 :strata cgd-seqnum
                :id cgd-id :agnostic t))
(def conditional-2
     (cox-model :times cgd-stop :entry cgd-start :status cgd-status
```

```
:x cgd-formula-2 :strata cgd-seqnum
:id cgd-id :agnostic t))
```

In the third analysis, often referred to as a *marginal* analysis or by the names of the researchers (Wei, Lin and Weissfeld, 1989) who proposed it, the first event, second event, third event and so on are treated as separate types of failure occuring in parallel. An individual is regarded as "at risk" for a second event even before the first. While this appears unreasonable there are good arguments in favour of this analysis in some cases. It is particularly suitable when there really are different parallel types of failure, but can also be useful for recurrent events of the same type. This analysis is again stratified on infection number but uses zero as the entry time for all infections. Again, the coefficients can be estimated separately or together for the different infection numbers (or any combination).

In this case we discard information about third and subsequent infections for simplicity. We must also construct imaginary records for the second infection in patients who did not have a first infection and so do not have a record. XLISP-Stat is not really suited to this sort of data management, and it would be more convenient in practice to set up the data before loading it.

45 ⟨*recurrent events* 45⟩≡

```
(wait)

(def cgd-data (col2row (list cgd-id cgd-seqnum cgd-stop
                             cgd-status cgd-trt )))

(def cgd-first-which (which (= 1 (seconds cgd-data))))
(def cgd-second-which (which (= 2 (seconds cgd-data))))

(def cgd-1 (select cgd-data cgd-first-which))
(def ids (firsts cgd-1))
(def cgd-2a (select cgd-data cgd-second-which))

(def has-second (firsts cgd-2a))
(def no-second
     (remove-if #'(lambda (xi) (eval (cons 'or (= xi has-second)))) ids))

(def cgd-2b
     (mapcar #'(lambda (xi)
                 (find xi cgd-1
                       :test #'(lambda(a b) (= a (elt b 0)))))
             no-second))
(def cgd-2bt (row2col cgd-2b))
(setf (select cgd-2bt 1) (repeat 2 (length no-second)))
(def cgd-2b (col2row cgd-2bt))

(def cgd1 (row2col (append cgd-1 cgd-2a cgd-2b)))
(def cgd1-id (elt cgd1 0))
(def cgd1-first (elt cgd1 1))
(def cgd1-stop (elt cgd1 2))
(def cgd1-status (elt cgd1 3))
(def cgd1-trt (elt cgd1 4))
```

```
(def cgd1-formula-1 (as-formula '((factor cgd1-trt))))
(def cgd1-formula-1-strat (as-formula '((factor cgd1-trt)
                                        (interaction (list cgd1-trt cgd1-first))
                                        )))

(def wlw-1
     (cox-model :times cgd1-stop :status cgd1-status
                :x cgd1-formula-1 :strata cgd1-first
                :id cgd1-id :agnostic t))
(def wlw-2
     (cox-model :times cgd1-stop  :status cgd1-status
                :x cgd1-formula-1-strat :strata cgd1-first
                :id cgd1-id :agnostic t))
```

3 Internal workings of the code

This section presents the details of coding and algorithms that were not given in the previous user-oriented section, beginning with the complete list of slot accessor methods and then giving the details of the fitting algorithm.

3.1 Layout of the Cox regression object

47 ⟨*cox regression object and methods* 47⟩≡

```
⟨tie-handling objects 31⟩
(defproto cox-regression-proto
  '(times entry status x formula beta id strata weights
         epsilon epsilon-dev count-limit verbose recycle
         eta loglike agnostic-covariance-matrix
         model-covariance-matrix tie-method recycle agnostic
         needs-computing cumulative-hazard
         martingales
         schoenfeld
         scores suppress-warnings
         predictor-names case-labels response-name
         statusint keyint entryint tieint comments time-dependent zbars
         paranoia
)
  '()
  *object*
  "Cox regression model")
⟨slot accessors and mutators 49⟩
⟨fitting 61⟩
⟨model summary methods 9⟩
⟨expected survival 18⟩
⟨diagnostics 23⟩

(defmeth cox-regression-proto :isnew (&key x times status
                                           entry
                                           id
                                           (formula nil)
                                           (ties *cox-default-ties*)
                                           pweights
                                           (verbose t)
                                           predictor-names
                                           response-name
                                           (recycle nil)
                                           case-labels
                                           init-beta
                                           count-limit
                                           (print t)
                                           print-summary
```

```
                                        statusint
                                        keyint
                                        agnostic
                                        entryint
                                        strata
                                        tieint
                                        comments
                                        (suppress-warnings nil)
                                        (compute t)
                                        (time-dependent nil)
                                        )
  (send self :x x)
  (send self :time-dependent time-dependent)
  (send self :times times)
  (send self :status status)
  (send self :entry entry)
  (send self :tie-method ties)
  (send self :weights (or pweights (repeat 1 (length times))))
  (send self :formula formula)
  (send self :strata strata)
  (send self :recycle recycle)
  (send self :verbose verbose)
  (send self :agnostic agnostic)
  (send self :suppress-warnings suppress-warnings)
  (when predictor-names (send self :predictor-names predictor-names))
  (when response-name (send self :response-name response-name))
  (send self :epsilon *cox-tolerance*)
  (send self :epsilon-dev  *cox-tolerance*)
  (send self :count-limit (or count-limit *cox-count-limit*))
  (send self :statusint statusint)
  (send self :keyint keyint)
  (send self :tieint tieint)
  (send self :entryint entryint)
  (send self :comments comments)
  (send self :id id)
  (cond (print (send self :display))
        (print-summary (send self :display-with-formula))
        (compute (send self :compute))
        )
)
```

49 ⟨*slot accessors and mutators* 49⟩≡

```
(defmeth cox-regression-proto :times (&optional (new nil set))
  "Message args: (&optional new)
 Sets or returns failure/censoring times"
  (when set
        (setf (slot-value 'times) new)
        (send self :needs-computing t))
  (slot-value 'times))


(defmeth cox-regression-proto :count-limit (&optional (new nil set))
  "Message args: (&optional new)
 Sets or returns iteration limit"
  (when set
        (setf (slot-value 'count-limit) new)
        (send self :needs-computing t))
  (slot-value 'count-limit))

(defmeth cox-regression-proto :entry (&optional (new nil set))
  "Message args: (&optional new)
 Sets or returns entry time"
  (when set
        (setf (slot-value 'entry) new)
        (send self :needs-computing t))
  (slot-value 'entry))

(defmeth cox-regression-proto :intercept () nil)

(defmeth cox-regression-proto :status (&optional (new nil set))
  "Message args: (&optional new nil set)
 Sets or returns failure indicator"
  (when set
        (setf (slot-value 'status) new)
        (send self :needs-computing t))
  (slot-value 'status))

(defmeth cox-regression-proto :id (&optional (new nil set))
"Message args: &optional (new nil set)
Sets or returns case id"
(when set
      (setf (slot-value 'id) id))
(slot-value 'id))

(defmeth cox-regression-proto :beta ()
  "Returns coefficient estimates"
  (slot-value 'beta))
```

```
(defmeth cox-regression-proto :keyint (&optional (new nil set))
"INTERNAL [ Message args: (&optional new)
  Sets or returns index from munged times into covariates]"
  (when set
        (setf (slot-value 'keyint) new)
        (send self :needs-computing t))
  (slot-value 'keyint))


(defmeth cox-regression-proto :statusint (&optional (new nil set))
  "INTERNAL [ Message args: (&optional new)
 Sets or returns failure status at munged times]"
  (when set
        (setf (slot-value 'statusint) new)
        (send self :needs-computing t))
  (slot-value 'statusint))

(defmeth cox-regression-proto :entryint (&optional (new nil set))
  "INTERNAL [Message args: (&optional new)
 Sets or returns entry status at munged times]"
  (when set
        (setf (slot-value 'entryint) new)
        (send self :needs-computing t))
  (slot-value 'entryint))

(defmeth cox-regression-proto :tieint (&optional (new nil set))
  "INTERNAL [Message args: (&optional new)
 Sets or returns tie status at munged times]"
  (when set
        (setf (slot-value 'tieint) new)
        (send self :needs-computing t))
  (slot-value 'tieint))

(defmeth cox-regression-proto :weights (&optional (new nil set))
  "Message args: (&optional new)
 Sets or returns case weights. These have no effect whatsoever."
  (when set
        (setf (slot-value 'weights) new)
        (send self :needs-computing t))
  (slot-value 'weights))

(defmeth cox-regression-proto :strata (&optional (new nil set))
  "Message args: (&optional new)
 Sets or returns stratum values"
  (when set
```

```
        (setf (slot-value 'strata) new)
        (send self :needs-computing t))
  (slot-value 'strata))

(defmeth cox-regression-proto :agnostic (&optional (new nil set))
  "Message args: (&optional new)
 Use agnostic covariance estimator"
  (when set
        (setf (slot-value 'agnostic) new)
        (send self :needs-computing t))
  (slot-value 'agnostic))


(defmeth cox-regression-proto :tie-method (&optional (new nil set))
  "Message args: (&optional new)
 Sets or returns tie-method"
  (when set
        (setf (slot-value 'tie-method) new)
        (send self :needs-computing t))
  (slot-value 'tie-method))

(defmeth cox-regression-proto :paranoia (&optional (new nil set))
   "Message args: (&optional new)"
   (when set
         (setf (slot-value 'paranoia) new)
         )
   (slot-value 'paranoia))


(defmeth cox-regression-proto :num-cases ()
  (length (unique (send self :id))))

(defmeth cox-regression-proto :x (&optional x)
  (if x
      (let ((x (cond
                ((matrixp x) x)
                ((vectorp x) (list x))
                ((and (consp x) (numberp (car x))) (list x))
                (t x))))
        (setf (slot-value 'x) (if (matrixp x) x (apply #'bind-columns x))))
  (slot-value 'x) )
)

(defmeth cox-regression-proto :formula (&optional (new nil set))
  "Message args: (&optional new)
 Sets or returns model formula"
```

```
  (when set
        (setf (slot-value 'formula) new)
        (send self :needs-computing t))
  (slot-value 'formula))

(defmeth cox-regression-proto :comments (&optional (new nil set))
  "Message args: (&optional new)
 Sets or returns model description text"
  (when set
        (setf (slot-value 'comments) new)
        )
  (slot-value 'comments))

(defmeth cox-regression-proto :epsilon (&optional (new nil set))
  "Message args: (&optional new)
 Sets or returns convergence criterion for beta"
  (when set
        (setf (slot-value 'epsilon) new)
        (send self :needs-computing t))
  (slot-value 'epsilon))

(defmeth cox-regression-proto :epsilon-dev (&optional (new nil set))
  "Message args: (&optional new)
 Sets or returns convergence criterion for loglikelihood"
  (when set
        (setf (slot-value 'epsilon-dev) new)
        (send self :needs-computing t))
  (slot-value 'epsilon-dev))

(defmeth cox-regression-proto :recycle (&optional (new nil set))
  "Message args: (&optional new)
 Sets or returns indicator for reusing old values when refitting"
  (when set
        (setf (slot-value 'recycle) new)
        (send self :needs-computing t))
  (slot-value 'recycle))

(defmeth cox-regression-proto :id (&optional (new nil set))
  "Message args: (&optional new)
 Sets or returns case identifier (important for agnostic
covariance estimates or to get single residuals per case
with time-dependent covariates)"
  (when set
        (setf (slot-value 'id) new)
        (send self :needs-computing t))
  (slot-value 'id))
```

```
(defmeth cox-regression-proto :covariance-matrix (&optional (new nil set))
  "Message args: (&optional new)
 Sets or returns covariance matrix"
  (if (send self :agnostic)
      (send self :agnostic-covariance-matrix)
    (send self :model-covariance-matrix))
  )

(defmeth cox-regression-proto :model-covariance-matrix (&optional (new nil set))
  "Message args: (&optional new)
 Sets or returns model-based covariance matrix"
  (when set
        (setf (slot-value 'model-covariance-matrix) new)
        )
  (slot-value 'model-covariance-matrix))

(defmeth cox-regression-proto :agnostic-covariance-matrix (&optional (new nil set))
  "Message args: (&optional new)
 Sets or returns agnostic (model-robust) covariance matrix"
  (when set
        (setf (slot-value 'agnostic-covariance-matrix) new)
        )
  (slot-value 'agnostic-covariance-matrix))

(defmeth cox-regression-proto :verbose (&optional (new nil set))
  "Message args: (&optional new)
 Sets or returns verboseness control (default is t to
report iteration information)"
  (when set
        (setf (slot-value 'verbose) new)
        (send self :needs-computing t))
  (slot-value 'verbose))

(defmeth cox-regression-proto :suppress-warnings (&optional (new nil set))
  (when set
        (setf (slot-value 'suppress-warnings) new)
        (send self :needs-computing t))
  (slot-value 'suppress-warnings))

(defmeth cox-regression-proto :loglike (&optional (new nil set))
  "Message args: (&optional new)
Sets or returns loglikelihood"
  (when set
        (setf (slot-value 'loglike) new)
        (send self :needs-computing t))
```

```
  (slot-value 'loglike))

(defmeth cox-regression-proto :needs-computing (&optional (new nil set))
  "Message args: (&optional new)
Flag: t if model needs updating"
  (when set
        (setf (slot-value 'needs-computing) new)
        )
  (slot-value 'needs-computing))

(defmeth cox-regression-proto :coef-standard-errors ()
(sqrt (diagonal (send self :covariance-matrix))))

(defmeth cox-regression-proto :model-coef-standard-errors ()
(sqrt (diagonal (send self :model-covariance-matrix))))

(defmeth cox-regression-proto :agnostic-coef-standard-errors ()
 (sqrt (diagonal (send self :agnostic-covariance-matrix))))

(defmeth cox-regression-proto :predictor-names (&optional (names nil set))
"Message args: (&optional (names nil set))
With no argument returns the predictor names. NAMES sets the names."
  (if set (setf (slot-value 'predictor-names) (mapcar #'string names)))
  (let* ((pfixed (array-dimension (send self :x) 1))
        (ptime (length (first (send self :time-dependent))))
        (p (+ pfixed ptime))
        (p-names (slot-value 'predictor-names)))
    (if (not (and p-names (= (length p-names) p)))
        (setf (slot-value 'predictor-names)
              (mapcar #'(lambda (a) (format nil "Variable ~a" a))
                      (iseq 0 (- p 1))))))
  (slot-value 'predictor-names))

(defmeth cox-regression-proto :response-name (&optional (name "Y" set))
"Message args: (&optional name)
With no argument returns the response name. NAME sets the name."
  (if set (setf (slot-value 'response-name) (if name (string name) "Y")))
  (slot-value 'response-name))

(defmeth cox-regression-proto :case-labels (&optional (labels nil set))
"Message args: (&optional labels)
With no argument returns the case-labels. LABELS sets the labels."
  (if set (setf (slot-value 'case-labels)
                (if labels
                    (mapcar #'string labels)
                    (mapcar #'(lambda (x) (format nil "~d" x))
```

```
                            (iseq 0 (- (send self :num-cases) 1))))))
  (slot-value 'case-labels))

(defmeth cox-regression-proto :set-beta (val)
"Args: beta
Set coefficient estimates: only done from within compute-step-beta
 and :initialize-search"
  (setf (slot-value 'beta) val)
  (slot-value 'beta)
)

(defmeth cox-regression-proto :fit-values ()
"Message args: ()
Returns the fitted linear predictor values for the model."
  (matmult (send self :x) (send self :beta)))

(defmeth cox-regression-proto :eta ()
"Message args: ()
Returns linear predictor values for current fit."
    (slot-value 'eta))

(defmeth cox-regression-proto :set-eta (&optional val)
  (if val
      (setf (slot-value 'eta) val)
      (setf (slot-value 'eta)
            (matmult (send self :x) (send self :beta))))
  (slot-value 'eta))

(defmeth cox-regression-proto :time-dependent (&optional (new nil set))
"Message args: &optional (new nil set)
Sets or returns the information on continuously time-dependent
covariates, consisting of a list containing a list of functions and a covariate matrix"
     (when set (setf (slot-value 'time-dependent) new))
     (slot-value 'time-dependent)
)

(defmeth cox-regression-proto :fit-hazard-ratio (&optional (eta (send self :eta)))
"Message args: (&optional (eta (send self :eta)))
Returns mean values for current or supplied ETA."
(if (null eta) (exp (send self :eta))  (exp eta)))

(defmeth cox-regression-proto :cumulative-hazard (&optional (new nil set))
 (when set
       (setf (slot-value 'cumulative-hazard) new)
       )
  (slot-value 'cumulative-hazard))
```

```
(defmeth cox-regression-proto :schoenfeld-residuals (&optional (new nil set))
 (when set
       (setf (slot-value 'schoenfeld) new)
       )
  (slot-value 'schoenfeld))

(defmeth cox-regression-proto :zbars (&optional (new nil set))
"Args: (new nil set)
Sets or returns a list whose elements are the time, stratum number, number at risk
in the stratum and the weighted mean covariate for each failure"
 (when set
       (setf (slot-value 'zbars) new)
       )
  (slot-value 'zbars))

(defmeth cox-regression-proto :scores (&optional (new nil set))
"Message ars: (&optional new)
Sets or returns martingale score residuals"
  (when set
       (setf (slot-value 'scores) new)
       )
  (slot-value 'scores))
```

3.2 Support functions

The first group of support functions is a set of graphical functions. `boxplot-factor]` performs a boxplot of one variable with
groups defined by the unique values of other variables, `[[plot-steps` and `add-steps` plot step functions, and `index-plot` plots a variable against the numbers $0, 1, 2, \ldots$.

The remaining functions perform various manipulations of sequences and matrices used in fitting and summarising the proportional hazards model.

57 ⟨*support functions* 57⟩≡

```
(defun index-plot (y &rest graphargs)
"Arguments: y &rest graphargs
Plots y against 1:(length y). Other arguments are passed to plot-points"
  (apply #'plot-points (append (list (iseq 1 (length y)) y) graphargs)))


(defun plot-steps (x y &rest graphargs &key (right t))
"Arguments: x y &rest graphargs &key (right t)
Plots a step function with the given step points.
Default is right-continuous, optionally left continuous"
(let* (
       (plot (apply #'plot-points (append (list x y) graphargs)))
       (xstep (if right 1 0))
       (ystep (if right 0 1))
       )
  (dotimes (i (- (length x) 1))
           (send plot :add-lines (select x (list i (+ i xstep) (+ i 1)))
                 (select y (list i (+ i ystep) (+ i 1))))
           )
  plot
  )
)

(defun add-steps (x y plot &key (right t))
"Args: x y plot &key (right t)
Adds a step function to plot, which must inherit from
scatterplot-proto"
(when (null (kind-of-p plot scatterplot-proto)) (error "Not a plot"))
(let* (
       (xstep (if right 1 0))
       (ystep (if right 0 1))
      )
  (send plot :add-points x y)
  (dotimes (i (- (length x) 1))
           (send plot :add-lines (select x (list i (+ i xstep) (+ i 1)))
                 (select y (list i (+ i ystep) (+ i 1))))
           )
```

```
  plot
  )
)

(defun boxplot-factor (x y)
"Args:x y
Plot boxplots of y for each level of x"
  (boxplot-x (unique x) (groups y x))
)

(defun cumprod (s) (accumulate #'* s))

(defun unique (x) (remove-duplicates x :test #'equalp))

(defun argselect (xs x) "returns indices where xs=s"
        (if (numberp x)
            (which (= x xs))
          (which (mapcar #'(lambda (xi) (equalp x xi)) xs))
          ))

(defun groups (y x)
"Args: y x
Break y into groups based on values in x"
  (let* ((xi (unique x))
         )
    (mapcar #'(lambda (i) (select y (argselect x i) )) xi)
    )
)

(defun zero-matrix (k) (* 0 (identity-matrix k)))

(defun one-matrix (k) (outer-product (repeat 1 k) (repeat 1 k)))

(defun firsts (x) (map-elements #'(lambda (xi) (elt xi 0)) x))
(defun seconds (x) (map-elements #'(lambda (xi) (elt xi 1)) x))
(defun thirds (x) (map-elements #'(lambda (xi) (elt xi 2)) x))
(defun fourths (x) (map-elements #'(lambda (xi) (elt xi 3)) x))
(defun nths (x n) (map-elements #'(lambda (xi) (elt xi (- n 1))) x))

(defun logg (x) (if (compound-data-p x)
                    (map-elements #'logg x)
                    (if (> x 0) (log x) 0)))

(defun col2row (x)
"Args: x
Converts a list of columns to a list of rows"
```

```
(row-list (apply #'bind-columns x)))

(defun row2col (x) "Args: x
Converts a list of rows to a list of columns"
(column-list (apply #'bind-rows x)))

(defun append-seq (&rest seqs)
  (apply #'append
         (mapcar #'(lambda (x) (coerce x 'list)) seqs)))

(defun flatten (listlist) (apply #'append listlist))

(defun list-of-seqs-p (x)
     (when (listp x) (when (sequencep (car x)) (numberp (car (car x))))))

(defun gtrp (x y) (if (numberp x) (> x y) (STRING> x y)))

;; used in the examples.
(defun wait ()
  (format t "--Press enter to continue--")
  (read-line))

;  the first four elements of a and b are time status entry stratum
; stratum could be a string

(defun after (a b)
  (if (equalp (elt a 3) (elt b 3))
        (if (= (elt a 0) (elt b 0))
            (if (= (elt a 2) (elt b 2))
                (< (elt a 1) (elt b 1))
              (> (elt a 2) (elt b 2))
              )
          (> (elt a 0) (elt b 0))
          )
    (gtrp (elt a 3) (elt b 3))
    )
)

(defun tied (a b) (if (and (= (elt a 0) (elt b 0))
                           (= 1 (elt a 1) (elt b 1))
                           (= (elt a 2) (elt b 2))
                           (equalp (elt a 3) (elt b 3)))
                      1 0))


(defun scatter-smooth (x y &rest graphargs &key (symmetric nil) (span .6))
```

```
"Arguments: x y &rest graphargs &key (symmetric nil) (span .6)
Draws a scatterplot with a lowess smoother. If symmetric is t uses the
two-iteration robust version (not good for skewed residuals).
Any extra arguments are passed to plot-points"
  (let* (
        (steps (if symmetric 2 1))
        (smooth (lowess x y :f span :steps steps))
        (plot (apply #'plot-points (append (list x y) graphargs)))
        )
    (send plot :add-lines (first smooth)  (second smooth) )
    plot
  )

)

(defun correlation (x y)
  (let* ((m (covariance-matrix x y)))
    (/ (aref m 0 1) (sqrt (prod (diagonal m))))
    )
)

(defun n+ (i j n) (if (null i) n (+ i j)))

(defun interpolate (xnew x y &key (right t))
"Interpolates the step function with corners at (x y) at the values
 in xnew. Default is a right-continuous step function like the
 Kaplan-Meier and Nelson-Aalen estimators"
(let* ((fudge (if right -1 0))
       (n (- (length y) 1)))
  (map-elements
   #'(lambda (xnewi) (elt y (n+ (position xnewi x :test #'<)  fudge n)))
   xnew)
  )
)
```

3.3 Parameter estimation and algorithms

A Newton–Raphson algorithm is used to maximise the loglikelihood, with iteration continuing up to a maximum specified by `:count-limit` or until the change in loglikelihood or the maximum relative change in the coefficients is less than `:epsilon`. As the partial loglikelihood is convex the Newton–Raphson algorithm should converge from any starting point, and the loglikelihood should increase at each step. A decrease in the loglikelihood indicates that rounding errors have become important, perhaps because of a strongly predictive covariate with some extreme values. This happens when the Mayo model is fitted with `bili` instead of `(log bili)`. If the loglikelihood decreases the last Newton–Raphson step is halved and a warning is printed. In addition, if the global variable `*cox-paranoia*` is `t`, step-halving is used whenever the ratio of the maximum and minimum values of $\exp(\beta' z)$ is greater than `(sqrt machine-epsilon)`, indicating a potentially serious loss of accuracy.

There are two versions of the algorithm. The first, using `:compute-tstep`, allows for time-dependence of covariates to be specified by arbitrary functions; the second, using `:compute-step` is substantially more efficient and assumes that covariates are constant for each observation, though step-function time-dependence is possible by having more than one record for each individual in the data set.

Both algorithms start at the last observation time and accumulate the risk set working backwards in time. At a failure time the loglikelihood, information matrix and score vector are updated, at a failure or censoring time the observation is added to the risk set and at an entry time it is removed from the risk set. The algorithms require that the data have been set up correctly, in order and with separate records for entry and exit times and an indicator for ties. This is done by the `(cox-model)` constructor function.

61 ⟨*fitting* 61⟩≡

```
(defmeth cox-regression-proto :initialize-search ()
  (let*
      ((np (+ (length (column-list (send self :x)))
              (length (first (send self :time-dependent))))))
    (send self :set-beta (repeat 0 np))) )

(defmeth cox-regression-proto :compute ()
  "Refits the model"
  (let* ((epsilon (send self :epsilon))
         (epsilon-dev (send self :epsilon-dev))
         (maxcount (send self :count-limit))
         (low-lim (* 2 (/ machine-epsilon epsilon)))
         (verbose (send self :verbose)))
    (unless (and (send self :beta) (send self :recycle))
            (send self :initialize-search))
    (do ((count 1 (+ count 1))
         (beta (send self :beta)  (send self :beta))
         (last-beta -1 beta)
         (dev  (- (/ machine-epsilon)) (send self :loglike))
         (last-dev  0 dev))
        ((or (> count maxcount)
             (< (max (abs (/ (- beta last-beta)
                             (pmax (abs last-beta) low-lim))))
                epsilon)
             (< (abs (- dev last-dev)) epsilon-dev))
```

```
          (when (and  (> count maxcount) (null (send self :suppress-warnings)))
                (format t
                        "~%~%Failed to converge in ~d iterations~%Treat results with caution ~%"
                        maxcount)))
          (send self :paranoia nil)
          (if (car (send self :time-dependent))
              (send self :compute-tstep)
            (send self :compute-step))
           (if (and (> count 1) (or (send self :paranoia) (< (- dev last-dev) (- epsilon-dev))))
            (progn
             (send self :set-beta (/ (+ beta last-beta) 2))
             (format t "Possible numerical instability: step-halving~%")))

          (if verbose
              (format t "Iteration ~d: log-likelihood = ~,6g~%"
                      count (send self :loglike)))
          )
      )
    (if (car (send self :time-dependent))
        (send self :compute-tmartingale)
      (send self :compute-martingale))
    (when  (send self :agnostic)
           (send self :compute-agnostic-covariance)
           )
    (send self :needs-computing nil)
)
```

⟨*One step with varying covariates* 66⟩
⟨*One step with constant covariates* 72⟩
⟨*martingale residual methods* 77⟩

The construction of the required data structure involves duplication and sorting of records that have both an entry and an exit time, and construction of an index variable to relate these records to the original data. Two extra variables are required, one containing 1 for failures and censorings and -1 for entries and one containing 1 for all but the first of a group of tied failures.

The data are ordered first by stratum then by time within stratum. At a given time entries precede failures and censorings come last. This ordering is encoded in the function `(after)`.

Apart from some input validation and this data rearrangement the `(cox-model)` function contains alternative calls to the constructor method `:new` for a model specified by a design matrix or by a model formula.

63 ⟨*internals of cox-model* 63⟩≡

```
(let* ((n (length times))
       (has-unique-id (= (length (remove-duplicates id)) (length id)))
       (comments (cond
                  ((and agnostic (not has-unique-id))
                   "Cox model with LWA/WLW agnostic covariance estimator")
                  ((and agnostic has-unique-id)
                   "Cox model with LW agnostic covariance estimator")
                  (t
                   "Standard Cox regression")
                  )
                 )
       (alldata (if entry
                    (col2row (list (append-seq entry times)
                                   (append-seq (repeat 0 n) status)
                                   (append-seq (repeat -1 n) (repeat 1 n))
                                   (append-seq strata strata) (repeat (iseq n) 2)))
                  (col2row (list times  status  (repeat 1 n) strata (iseq n)))
                  ))
       (rtemp  (sort (copy-list alldata) #'after))
       (temp (row2col rtemp))
       (keyint (select temp 4))
       (statusint (select temp 1))
       (entryint (select temp 2))
       (ntimes (length keyint))
       (timexlist (cond
                   ((null time-x)
                    nil)
                   ((matrixp time-x)
                       (row-list time-x))
                   ((and (sequencep time-x) (numberp (elt time-x 0)))
                    (row-list (bind-columns time-x)))
                   ((and (sequencep time-x) (sequencep (elt time-x 0)))
                    (row-list (apply #'bind-columns time-x)))
                   (t
                    (error "Wrong sort of :time-x"))
                   ))
```

```
        ;; tieint =1 if this entry is tied with the previous one, else 0
        (tieint (append
                 (mapcar #'tied (select rtemp (iseq 0 (- ntimes 2)))
                         (select rtemp (iseq 1 (- ntimes 1))))
                 '(0)))
        )
    (cond ((objectp x)
           (send cox-regression-proto :new :x (send x :design-matrix)
                 :times times :status status :entry entry
                 :predictor-names
                 (if has-names predictor-names (send x :name-list))
                 :init-beta init-beta
                 :count-limit count-limit
                 :weights weights
                 :formula x
                 :print nil
                 :print-summary print
                 :agnostic agnostic
                 :verbose verbose
                 :keyint keyint
                 :strata strata
                 :statusint statusint
                 :entryint entryint
                 :tieint tieint
                 :ties ties
                 :reponse-name response-name
                 :comments comments
                 :suppress-warnings suppress-warnings
                 :time-dependent (list time-funs timexlist)
                 :id id
                 ))
          (t
           (send cox-regression-proto :new :x x
                 :times times :status status :entry entry
                 :predictor-names predictor-names
                 :init-beta init-beta
                 :count-limit count-limit
                 :weights weights
                 :formula x
                 :print print
                 :print-summary nil
                 :agnostic agnostic
                 :verbose verbose
                 :strata strata
                 :keyint keyint
                 :statusint statusint
```

```
                :entryint entryint
                :tieint tieint
                :ties ties
                :reponse-name response-name
                :comments comments
                :suppress-warnings suppress-warnings
                :time-dependent (list time-funs timexlist)
                   :id id
                   )
         )
        )
   )
```

The code proceeds stratum-by-stratum and backwards through time within a stratum, accumulating the risk set and the loglikelihood, score and information matrix. An observation is classified as a tied failure, the last of a set of tied failures, an untied failure, a censoring, or an entry time, and is used to update the risk set.

Computations are postponed on tied failures until the whole set of ties has been accumulated, at which time the `tie-method` is called to update the loglikelihood, score, and information matrix and to compute the Schoenfeld residuals. For an untied failure these calculations are done immediately. For a censoring or entry time there is no contribution to the partial likelihood and no computations are necessary.

The computation for a failure involves adding up the risk score $\exp(\beta' z)$ and its derivatives over the risk set at that time. This is implemented in an inner loop.

66 ⟨*One step with varying covariates* 66⟩≡

```
(defmeth cox-regression-proto :compute-tstep
  ()
"Does one step of Newton-Raphson algorithm for models with
continuously time-dependent covariates"
(let* (
       (keyint (send self :keyint))
       (statusint (send self :statusint))
       (entryint (send self :entryint))
       (tieint (send self :tieint))
       (times (send self :times))
       (strata (send self :strata))
       (stratlist (unique strata))
       (x-list (row-list (send self :x)))
       (riskmin 1)
       (riskmax 1)
       (oldbeta (send self :beta))
       (timecovs (send self :time-dependent))
       (timefuns (first timecovs))
       (timexs (second timecovs))
       (pfixed (length (first x-list)))
       (ptime (if timecovs (length (first timecovs)) 0))
       (p (+ pfixed ptime))
       (nz (length x-list))
       (nt (length keyint))
       (dcumhaz nil)
       (schoenfelds nil)
       (loglike 0)
       (score (repeat 0 p))
       (varscore (matrix (list p p) (repeat 0 (* p p))))
       (current-ties nil)
       (tie-method (send self :tie-method))
       (last-stratum (elt strata (elt keyint 0)))
       (this-stratum (elt strata (elt keyint 0)))
       (zbars nil)
       (riskset nil)
```

```
(junk (dotimes (i nt)
                (setf this-stratum (elt strata (elt keyint i)))
                (cond ((equalp this-stratum last-stratum)
                       (setf last-stratum this-stratum
                             ))
                      (t
                       (setf riskset nil
                             last-stratum this-stratum
                             ))
                      )
                (cond ((= (elt tieint i) 1) ;;one of a set of ties
                       (setf current-ties
                             (cons (elt keyint i) current-ties))
                       )
                      (current-ties ;; last of a tied set of failures
                       (unless (= (elt statusint i) 1)
                               (error "Can't happen"))
                       (setf current-ties
                             (cons (elt keyint i) current-ties))
                         ;;; recalculate nS0, nS1, nS2 first
                       (setf nS0 0)
                       (setf nS1 (repeat 0 p))
                       (setf nS2 (zero-matrix p))
                       (setf riskset (append current-ties riskset))
                       (dolist (this-obs riskset)
                               (let* ((timez (mapcar
                                              #'(lambda (f)
                                                  (funcall f
                                                           (elt timexs this-obs)
                                                           (elt times (elt keyint i))))
                                              timefuns))
                                      (z (append-seq
                                          (select x-list this-obs)
                                          timez))
                                      (ebtz (exp (inner-product z oldbeta)))
                                      (setf riskmin (min riskmin ebtz))
                                      (setf riskmax (max riskmax ebtz))
                                      )
                                 (setf nS0 (+ nS0 ebtz))
                                 (setf nS1 (+ nS1 (* z ebtz)))
                                 (setf nS2 (+ nS2 (* ebtz (outer-product z z))))
                                 )
                               )
                       (let* ((timexlist (select timexs current-ties))
                              (timezlist (mapcar
                                          #'(lambda (keyi)
```

```
                                (mapcar
                                 #'(lambda (f) (funcall f
                                                        (elt timexs keyi)
                                                        (elt times keyi)))
                                 timefuns))
                            current-ties))
                 (zlist (row-list
                         (bind-columns
                          (apply #'bind-rows (select x-list current-ties))
                          (apply #'bind-rows timezlist))))
                 (temp (send tie-method
                             :increment zlist  oldbeta nS0 nS1 nS2))
                 (thistime (select times (select keyint i)))
                 )
           (setf score (+ score (first temp)))
           (setf varscore (+ varscore (second temp)))
           (setf loglike (+ loglike
                            (send tie-method
                                  :dloglike zlist oldbeta ns0)))
           (setf dcumhaz
                 (cons
                  (list thistime
                        (send tie-method
                              :dLambda  zlist oldbeta nS0)
                        this-stratum)
                  dcumhaz))
           (setf schoenfelds
                 (append
                  (mapcar #'(lambda (si)
                              (list thistime
                                    si
                                    this-stratum))
                            (send tie-method
                                  :schoenfeld zlist oldbeta ns0 ns1))
                  schoenfelds))
           (setf zbars (cons
                        (list thistime
                              this-stratum
                              (length riskset)
                              (repeat (/ nS1 nS0)
                                      (length current-ties))
                              )
                        zbars))
           (setf current-ties nil)
           )
         )
```

```
((= (elt statusint i) 1)  ;; untied failure
 (setf riskset (cons (elt keyint i) riskset))
   ;;; recalculate nS0, nS1, nS2 first
 (setf nS0 0)
 (setf nS1 (repeat 0 p))
 (setf nS2 (zero-matrix p))
 (dolist (this-obs  riskset)
         (let* ((timez (mapcar
                        #'(lambda (f)
                            (funcall f
                                     (elt timexs this-obs)
                                     (elt times (elt keyint i))))
                        timefuns))
                (z (append-seq
                    (select x-list this-obs)
                    timez))
                (ebtz (exp (inner-product z oldbeta)))
                (setf riskmin (min riskmin ebtz))
                (setf riskmax (max riskmax ebtz))
                )
           (setf nS0 (+ nS0 ebtz))
           (setf nS1 (+ nS1 (* z ebtz)))
           (setf nS2 (+ nS2 (* ebtz (outer-product z z))))
           )
         )

 (let* ((timez (mapcar
                #'(lambda (f)
                    (funcall f
                             (elt timexs (elt keyint i))
                             (elt times (elt keyint i))))
                timefuns))
        (z (append-seq
            (select x-list (select keyint i))
            timez))
        (btz (inner-product z oldbeta))
        (ebtz (exp btz))
        (thistime (select times (select keyint i)))
        )
   (setf current-ties nil)
   (setf score (+ score (- z (/ nS1 nS0))))
   (setf varscore (+ varscore
                     (/ nS2 nS0)
                     (/ (outer-product (- nS1) nS1)
                        (* nS0 nS0))))
   (setf dcumhaz (cons
```

```
                                    (list thistime (/ ns0) this-stratum)
                                    dcumhaz))
                          (setf schoenfelds (cons
                                    (list thistime
                                          (- z (/ nS1 nS0))
                                          this-stratum)
                                    schoenfelds))
                          (setf zbars (cons
                                    (list thistime
                                          this-stratum
                                          (length riskset)
                                          (/ nS1 nS0))
                                    zbars))
                          (setf loglike (+ loglike btz (- (log nS0))))
                          )
                         )
                        ((= (elt entryint i) 1) ;; censoring
                         (setf riskset
                               (cons (elt keyint i) riskset))
                         )
                        ((= (elt entryint i) -1) ;; entry
                         (if (position (elt keyint i) riskset)
                             (setf riskset (remove (elt keyint i) riskset))
                           (error "Entry time with no corresponding exit time"))
                         )
                        (t (error "Can't happen in :compute-step"))
                        )
                  ))
      (invL (inverse (first (chol-decomp varscore))))
      (inv (matmult (transpose invL) invL))
      (cumhaz (map-elements
               #'(lambda (stratum)
                   (let* ((these (select zbars
                                         (argselect (seconds zbars) stratum) )))
                     (if these
                         (list stratum
                               (append-seq #(0) (firsts these))
                               (append-seq #(0) (cumsum (seconds these))))
                       (list stratum '(0 0) '(0 0)))))
                 stratlist))
      (rotzbars (map-elements
                 #'(lambda (stratum)
                   (let* ((these (select zbars
                                         (argselect (seconds zbars) stratum) )))
                     (if these
                         (list stratum
```

```
                                    (firsts these)
                                    (fourths these)
                                    (thirds these))
                                (list stratum '(0) (list (repeat 0 p)) '(0)))))
                            stratlist))
          )
    (send self :set-beta (+ oldbeta (matmult score inv)))
    (send self :model-covariance-matrix inv )
    (send self :loglike loglike)
    (send self :cumulative-hazard cumhaz)
    (send self :schoenfeld-residuals schoenfelds)
    (send self :zbars rotzbars)
    (when (and *cox-paranoia* ( < (/ riskmin riskmax) (sqrt machine-epsilon)))
    (send self :paranoia t))
    )
  )
```

When covariates do not change with time a substantial saving in effort is possible as the weighted sums $S^{(i)}(t)$ required to calculate the loglikelihood, score vector and information matrix can be accumulated over time without storing the whole risk set. When an observation is added to the risk set its contribution is added to $S^{(i)}$; when it is removed its contribution is subtracted. This procedure removes the inner loop of the previous version and reduces the number of operations required from $O(n^2)$ to $O(n)$, at the cost of inducing a potential numerical instability by repeated addition and subtraction. Given the large number of digits accuracy used in most modern floating-point systems this instability is likely to be a rare occurence and to occur only when there is substantial left-truncation. If it is a concern the slower and more precise algorithm may be used.

72 ⟨*One step with constant covariates* 72⟩≡

```
(defmeth cox-regression-proto :compute-step ()
"Does one step of Newton-Raphson algorithm"
(let* (
       (keyint (send self :keyint))
       (statusint (send self :statusint))
       (entryint (send self :entryint))
       (tieint (send self :tieint))
       (times (send self :times))
       (strata (send self :strata))
       (stratlist (unique strata))
       (x-list (row-list (send self :x)))
       (riskmin 1)
       (riskmax 1)
       (oldbeta (send self :beta))
       (p (length (first x-list)))
       (nz (length x-list))
       (nt (length keyint))
       (dcumhaz nil)
       (schoenfelds nil)
       (nS0 0)
       (nrisk 0)
       (loglike 0)
       (nS1 (repeat 0 p))
       (nS2 (matrix (list p p) (repeat 0 (* p p))))
       (score (copy-seq nS1))
       (varscore (copy-array nS2))
       (current-ties 0)
       (zbars nil)
       (tie-method (send self :tie-method))
       (last-stratum (elt strata (elt keyint 0)))
       (this-stratum (elt strata (elt keyint 0)))
       (junk (dotimes (i nt)
                      ;; work backwards in time, accumulating information as we go
                      (setf this-stratum (elt strata (elt keyint i)))
                      (cond ((equalp this-stratum last-stratum)
                             (setf last-stratum this-stratum
```

```
                ))
              (t
               (setf nrisk 0
                     nS0 0
                     nS1 (repeat 0 p)
                     nS2 (matrix (list p p) (repeat 0 (* p p)))
                     last-stratum this-stratum
                     ))
              )
            (cond ((= (elt tieint i) 1)  ; tied failure
                   (let* (
                          (z (select x-list (select keyint i)))
                          (ebtz (exp (inner-product z oldbeta)))
                          )
                     (setf riskmin (min riskmin ebtz))
                     (setf riskmax (max riskmax ebtz))
                     (setf nS0 (+ nS0 ebtz))
                     (setf nS1 (+ nS1 (* z ebtz)))
                     (setf nS2 (+ nS2 (* ebtz (outer-product z z))))
                     (setf current-ties (+ current-ties 1))
                     (setf nrisk (+ nrisk 1))
                     )
                   )
                  ((> current-ties 0)  ; last of tied set of failures
                   (let* (
                          (z (select x-list (select keyint i)))
                          (ebtz (exp (inner-product z oldbeta)))
                          )
                     (setf riskmin (min riskmin ebtz))
                     (setf riskmax (max riskmax ebtz))
                     (setf nS0 (+ nS0 ebtz))
                     (setf nS1 (+ nS1 (* z ebtz)))
                     (setf nS2 (+ nS2 (* ebtz (outer-product z z))))
                     (setf nrisk (+ 1 nrisk))
                     (setf current-ties (+ current-ties 1)))
                   (let* ((keylist (select keyint (- i (iseq  current-ties))))
                          (zlist (select x-list keylist))
                          (temp (send tie-method
                                      :increment zlist  oldbeta nS0 nS1 nS2))
                          (thistime (select times (select keyint i)))
                          )
                     (setf score (+ score (first temp)))
                     (setf varscore (+ varscore (second temp)))
                     (setf loglike (+ loglike
                                      (send tie-method :dloglike
                                            (select x-list keylist)
```

```
                              oldbeta ns0)))
      (setf dcumhaz (cons
                     (list thistime
                           (send tie-method :dLambda
                                 zlist
                                 oldbeta
                                 nS0)
                           this-stratum)
                     dcumhaz))
      (setf schoenfelds (append
                         (mapcar #'(lambda (si)
                                     (list thistime
                                           si
                                           this-stratum))
                                 (send tie-method :schoenfeld
                                       zlist
                                       oldbeta
                                       ns0
                                       ns1))
                         schoenfelds))
      (setf zbars (cons
                   (repeat (list thistime
                                 this-stratum
                                 nrisk
                                 (/ nS1 nS0))
                           current-ties)
                   zbars))
      (setf current-ties 0)
      )
     )
    ((= (elt statusint i) 1)  ; untied failure
     (let* ((z (select x-list (select keyint i)))
            (btz (inner-product z oldbeta))
            (ebtz (exp btz))
            (thistime (select times (select keyint i)))
            )
      (setf riskmin (min riskmin ebtz))
      (setf riskmax (max riskmax ebtz))
      (setf nS0 (+ nS0 ebtz))
      (setf nS1 (+ nS1 (* z ebtz)))
      (setf nS2 (+ nS2 (* ebtz (outer-product z z))))
      (setf current-ties 0)
      (setf nrisk (+ nrisk 1))
      (setf score (+ score (- z (/ nS1 nS0))))
      (setf varscore (+ varscore
                        (/ nS2 nS0)
```

```
                                        (/ (outer-product (- nS1) nS1)
                                           (* nS0 nS0))))
                           (setf dcumhaz (cons (list thistime
                                                     (/ ns0)
                                                     this-stratum)
                                               dcumhaz))
                           (setf schoenfelds (cons (list thistime
                                                         (- z (/ nS1 nS0))
                                                         this-stratum)
                                                   schoenfelds))
                           (setf zbars (cons (list thistime
                                                   this-stratum
                                                   nrisk
                                                   (/ nS1 nS0))
                                             zbars))
                           (setf loglike (+ loglike btz (- (log nS0))))
                           )
                         )
                         ((= (elt entryint i) 1) ;censoring
                          (let* ((z (select x-list (select keyint i)))
                                 (ebtz (exp (inner-product z oldbeta)))
                                 )
                           (setf riskmin (min riskmin ebtz))
                           (setf riskmax (max riskmax ebtz))
                           (setf nS0 (+ nS0 ebtz))
                           (setf nS1 (+ nS1 (* z ebtz)))
                           (setf nS2 (+ nS2 (* ebtz (outer-product z z))))
                           (setf nrisk (+ nrisk 1))
                           )
                          )
                         ((= (elt entryint i) -1) ;left truncation
                          (let* ((z (select x-list (select keyint i)))
                                 (ebtz (exp (inner-product z oldbeta)))
                                )
                           (setf riskmin (min riskmin ebtz))
                           (setf riskmax (max riskmax ebtz))
                           (setf nrisk (- nrisk 1))
                           (setf nS0 (- nS0 ebtz))
                           (setf nS1 (- nS1 (* z ebtz)))
                           (setf nS2 (- nS2 (* ebtz (outer-product z z))))
                           )
                          )
                         (t (error "Can't happen in :compute-step"))
                         )
                   ))
    (invL (inverse (first (chol-decomp varscore))))
```

```
        (inv (matmult (transpose invL) invL))
        (cumhaz (map-elements
                 #'(lambda (stratum)
                     (let* ((these (select dcumhaz
                                           (argselect (thirds dcumhaz) stratum)))
                            )
                       (if these
                           (list stratum
                                 (append-seq  #(0) (firsts these))
                                 (append-seq  #(0) (cumsum (seconds these))))
                         (list stratum '(0 0) '(0 0))
                         )
                       ))
                 stratlist))
        (rotzbars (map-elements
                    #'(lambda (stratum)
                        (let* ((these (select zbars
                                              (argselect (seconds zbars) stratum) )))
                          (if these
                              (list stratum
                                    (firsts these)
                                    (fourths these)
                                    (thirds these))
                            (list stratum '(0) (list (repeat 0 p))  '(0))
                            )
                          ))
                    stratlist))
        )
    (send self :set-beta (+ oldbeta (matmult score inv)))
    (send self :model-covariance-matrix inv )
    (send self :loglike loglike)
    (send self :cumulative-hazard cumhaz)
    (send self :schoenfeld-residuals schoenfelds)
    (send self :zbars rotzbars)
    (when (and *cox-paranoia* ( < (/ riskmin riskmax) (sqrt machine-epsilon)))
      (send self :paranoia t))
    )
  )
```

3.4 Martingale residuals and Agnostic covariances

Much survival analysis theory is based on the counting process formulation in which the martingale, the difference between observed and expected numbers of events, plays a key role. The martingale residuals are the differences between observed and expected numbers of events for each individual evaluated at the end of time. They are useful in their own right as diagnostics and are required to calculate the score residuals.

The score residuals, as described in section 2.5 are used to compute influence diagnostics ($\Delta\beta$s) and the model-robust or agnostic covariance matrix. At the moment these quantities are not available under arbitrary time-dependence.

77 ⟨*martingale residual methods* 77⟩≡

```
(provide "martingale")

(defun outer-square (seq) (outer-product seq seq))

(defmeth cox-regression-proto :compute-martingale ()
"Martingale and score residuals collapsed over entry/exit but not id"
(let* (
       (keyint (send self :keyint))
       (statusint (send self :statusint))
       (entryint (send self :entryint))
       (tieint (send self :tieint))
       (times (send self :times))
       (entrytimes (send self :entry))
       (strata (send self :strata))
       (stratlist (unique strata))
       (x-list (row-list (send self :x)))
       (p (length (first x-list)))
       (nz (length x-list))
       (nt (length keyint))
       (cumhaz (send self :cumulative-hazard))
       (zbars (send self :zbars))
       (id (send self :id))
       (beta (send self :beta))
       (dcumhaz  (map-elements
                  #'(lambda (stratum)  (list (first stratum)
                                             (second stratum)
                                             (difference (third stratum))))
                  cumhaz))
       (zbardL (map-elements  #'(lambda (hazs zbars)
                                  (list (first hazs)
                                        (second hazs)
                                        (* (third hazs) (third zbars))))
                              dcumhaz
                              zbars))
       (intzbardL (map-elements  #'(lambda (stratum)
                                     (list (first stratum)
```

```
                                    (second stratum)
                                    (cumsum (third stratum))))
                            zbardL))
        (rval (row-list (bind-columns (repeat 0 nz) (* 0 (send self :x)))))
        (fn #'(lambda (caseno)
                (let* ((key (elt keyint caseno))
                       (idno (elt id key))
                       (stratno (elt strata key))
                       (status (elt statusint caseno))
                       (entry (elt entryint caseno))
                       (thistime (if (= entry 1)
                                     (elt times key)
                                   (elt entrytimes key)))
                       (zi (elt x-list key))
                       (risk (exp (inner-product zi beta)))
                       (whichtminus (position
                                     thistime
                                     (second (find stratno
                                                   zbars
                                                   :test #'(lambda (x y)
                                                             (equalp  x (elt y 0)))))
                                     :test #'>=
                                     :from-end t))
                       (zbarti (if whichtminus
                                   (elt (third (find
                                                stratno
                                                zbars
                                                :test #'(lambda (x y)
                                                          (equalp x (elt y 0)))))
                                        whichtminus)
                                 (* 0 zi)))
                       (Lti (* risk (if whichtminus
                                        (elt (third (find
                                                     stratno
                                                     cumhaz
                                                     :test #'(lambda (x y)
                                                               (equalp  x (elt y 0)))))
                                             (+ 1 whichtminus)) 0)))
                       (intzbardLti (* risk
                                       (if whichtminus
                                           (elt (third (find
                                                        stratno
                                                        intzbardL
                                                        :test #'(lambda (x y)
                                                                  (equalp x (elt y 0)))))
                                                whichtminus)
```

```
                                    0)))
                      (Mti (cons
                            (- status Lti)
                            (coerce (+
                                     (* status (- zi zbarti))
                                     intzbardLti
                                     (- (* zi Lti)))
                                    'list)))
                      )
                    (setf (select rval key) (+ (select rval key) (* entry Mti)))
                    )
                   t))
        (junk (map-elements fn (iseq nt)))
        (mgales (column-list (apply #'bind-rows rval)))
        )
   (send self :martingales (first mgales))
   (send self :scores (apply #'bind-columns (cdr mgales)))
   )
t
)


(defmeth cox-regression-proto :compute-tmartingale ()
"Martingale and score residuals collapsed over id for
 general time-dependence"
(format t
        "Warning: martingale residuals not yet available for general time-dependence\n")
)


(defmeth cox-regression-proto :compute-agnostic-covariance ()
"INTERNAL: compute and set appropriate agnostic covariance matrix"
(let* ((id (send self :id))
       (unique-id (coerce (unique id) 'list))
       (scores (or
                (row-list (send self :scores))
                (error "Can't do agnostic covariance - no martingale residuals")))
       (sumsq (cond ( (= (length id) (length unique-id))
                      (apply #'+
                             (mapcar #'outer-square scores)))
                    (t
                     (apply #'+
                            (mapcar #'outer-square
                                    (mapcar #'(lambda (i)
                                                (apply #'+
                                                       (select scores
```

```
                                                                    (argselect id i))))
                                                unique-id))))
                    ))
        (modelv (send self :model-covariance-matrix))
        )
    (send self :agnostic-covariance-matrix (matmult modelv sumsq modelv))
  ))
```

4 Model formula objects

The model formula objects and their supporting functions provide an easier way to specify design matrices, especially when factor variables and interactions are required. They also provide facilities for Wald tests for an entire factor or interaction term. The main difficulty in providing these model formula facilities is the lack of a 'lazy evaluation' mechanism such as that in S. This means that a function can access only the values and not the names of its formal parameters. In order to construct useful variables names the function needs access to unevaluated arguments. This is done by requiring the argument to be quoted.

The function `(as-formula)` constructs a model formula object. Its argument is a quoted list where `(term x)` specifies that `x` is to be entered as a column in the design matrix (a 'linear' or 'metric' variable), `(factor y)` means that `y` is to be coded as a set of indicator variables using treatment contrasts, and `(interaction (list x y z) :is-factor (list nil t t))` specifies a three way interaction between the factor variable `x` and the metric variables `y` and `z`, *i.e.*, a set of columns with values $y \times z$ or 0 for each value of x. If all the variables in an interaction are factors the `:is-factor` argument may be omitted. A factor variable may contain strings or numbers, but not symbols.

The `model-formula-proto` has methods `:design-matrix` to return a design matrix, `:name-list` to give names for the columns of the design matrix, `:block-names` to give names for the terms in the original model formula and `:block-indices` to specify which columns of the design matrix correspond to which model formula terms.

The main computation is carried out by the `(formula)` function, which can be used without the model formula objects. It takes a quoted model formula and returns a list containing the design matrix, name list, block names and block indices. It in turn uses `(eval)` to call the `(term)`, `(factor)` and `(interaction)` functions, which produce the design matrix columns and names for one model formula term.

The function `(formula-display)` shows how the model formula can be used to display a fitted model. It is preferable to write a `:display-with-formula` method for the regression model as `(formula-display)` has no way to check that its arguments match each other in anything but dimensions. The `cox-regression-proto` has such a `:display-with-formula` method. These call the `(block-test)` function to perform Wald tests for each term in the model formula.

81 ⟨*model formula object and methods* 81⟩≡

```
(defun term (x &key namebase)
  "A metric predictor variable"
  (if namebase
      (list (bind-columns x)  namebase))
  (bind-columns x)
  )

(defun factor (x &key namebase )
"Defines treatment contrast matrix for x and optionally
a set of names based on namebase"
  (let* (
         (xlist (sort-data (remove-duplicates (coerce x 'list) :test #'equalp)))
         (p (- (length xlist) 1))
         (rows (row-list (select (identity-matrix (+ p 1))
                                 (iseq 0 p) (iseq 1 p))))
         (decoder (mapcar #'cons xlist rows))
```

```
          (xmatrix (apply #'bind-rows
                          (coerce (map-elements #'(lambda (xx)
                                      (cdr (assoc xx decoder
                                                  :test #'equalp)))
                                    x) 'list)))
          (xnames (if (null  namebase)
                      nil
                    (cdr (mapcar #'(lambda (xx)
                                     (write-to-string (list namebase xx)
                                                      :escape nil))
                                 xlist))))
     )
     (if (null namebase)
         xmatrix
       (list xmatrix xnames))
     )
  )

  (defun interaction ( xs &key (is-factor (repeat t (length xs))) namebases )
  "Interactions of any order for categorical and continuous variables"
    (let*  ( (n (length (first xs)))
             (zlist nil)
             (znames nil)
             (p (length xs))
             (names (if (null namebases)
                        (repeat "." p)
                      namebases))
             (junk (dotimes (i p)
                           (let* (
                                  (thisz  (if (elt is-factor i)
                                              (factor (elt xs i)
                                                      :namebase (elt names i))
                                            (list (bind-columns (elt xs i))
                                                  (list (elt names i)))))
                                  )
                             (if (null zlist)
                                 (setf zlist (column-list (first thisz)))
                               (setf zlist
                                     (apply #'append
                                            (mapcar #'(lambda (xx)
                                                        (mapcar #'(lambda (yy)
                                                                    (*  xx yy))
                                                                (column-list
                                                                 (first thisz))))
                                                    zlist))))
                             (if (null znames)
```

```
                              (setf znames (second thisz))
                             (setf znames
                                   (apply #'append
                                          (mapcar #'(lambda (xx)
                                                      (mapcar #'(lambda (yy)
                                                                  (write-to-string
                                                                   (list xx yy)
                                                                   :escape nil))
                                                              (second thisz)))
                                                  znames))))
                            )
                           )
                 )
          (zmat (apply #'bind-columns zlist))
          )
    (if (null namebases)
        zmat
      (list zmat znames))
))

(defun design (varlist) (apply #'bind-columns (mapcar #'first varlist)))

(defun names (varlist) (apply #'append (mapcar #'second varlist)))

(defun col-count (matlist)
  "adds up columns"
  (apply #'sum (mapcar #'(lambda (x)
                           (cond ((matrixp x)
                                  (second (array-dimensions x)))
                                 (t 1)))
                       matlist)))

(defun formula (model-formula)
"Args: model-formula
Calculates a design matrix, variable names and information for running
(block-test) on the fitted model. The argument is  a quoted list of metric
variables, factor variables and interactions. Metric variables are
specified by (term x), factor variables by (factor a) and interactions
 by (interaction (list a b x) :is-factor (list t t nil)). "
(let* (
       (design-matrix nil)
       (name-list nil)
       (block-names nil)
       (block-indices nil)
       (i 0)
       (junk (dolist (x model-formula)
```

```
(setf i (if (null design-matrix)
            0
          (col-count design-matrix)))
(cond
 ((eql (car x) 'term)
  (let* ((namepos (position ':namebase x))
         )
    (setf
     design-matrix (cons (eval (second x)) design-matrix)
     name-list (cons (if (null (find ':namebase x))
                         (list (write-to-string (second x)))
                       (list (fourth x))) name-list)
     block-names (cons (write-to-string
                        (if (null namepos)
                            (second x)
                          (elt x (+ 1 namepos))))
                       block-names)
     block-indices (cons (list i) block-indices)
     )
    )
  )
  ((eql (car x) 'factor)
   (let* ((namepos (position ':namebase x))
          (factorx (eval
                    (if (null namepos)
                        (append x
                                (list :namebase
                                      (write-to-string (second x))))
                      x)))
          )
     (setf
      design-matrix (cons (first factorx) design-matrix)
      name-list (cons (second factorx) name-list)
      block-names (cons (write-to-string
                         (if (null namepos)
                             (second x)
                           (elt x (+ 1 namepos))))
                        block-names)
      block-indices (cons
                     (iseq i (- (col-count design-matrix) 1))
                     block-indices)
      )
     )
   )
  ((eql (car x) 'interaction)
   (let* (
```

```
                                (namepos (position ':namebases x))
                                (interx (eval
                                         (if (null namepos)
                                             (append x
                                                     (list :namebases
                                                           (cons 'list
                                                                 (mapcar
                                                                  #'write-to-string
                                                                  (cdr (second x))))))
                                           x)))
                                )
                               (setf
                                design-matrix (cons (first interx) design-matrix)
                                name-list (cons (second interx) name-list)
                                block-names (cons (write-to-string
                                                   (cons 'interaction
                                                         (if (null namepos)
                                                             (cdr (second x))
                                                           (cdr (elt x (+ 1 namepos))))
                                                         )
                                                   :escape nil)
                                                  block-names)
                                block-indices (cons
                                               (iseq i (- (col-count design-matrix) 1))
                                               block-indices)
                                )
                               )
                              )
                             (t (error "I don't understand ~s" x))
                             )
                            )
                    )
               )
         )
    (list (apply #'bind-columns design-matrix)
          (apply #'append name-list)
          block-names
          (- (- (col-count design-matrix) 1)
             block-indices))
    )
  )

(defun block-test (index beta covmat &key blockname names (block-only nil))
  (let* (
         (subbeta (select beta index))
         (subcov  (select covmat index index))
         (waldchisq (inner-product subbeta (matmult (inverse subcov) subbeta)))
```

```
          (waldp (- 1 (chisq-cdf waldchisq (length index))))
          (subse (sqrt (diagonal subcov)))
          (subz (/ subbeta subse))
          (subp (* 2 (- 1 (normal-cdf (abs subz)))))
          (nn (if (null names)
                  (repeat "" (length index))
                (select names index)))
          (blockn (if (null blockname)
                      "block"
                    blockname))
          )
    (format t "~a~20t~13,5g~35t~,4f~%" blockn waldchisq waldp)
    (unless block-only
            (format t "~5t Variable~25t Estimate~40t Std.Err.~%")
            (dolist (i (iseq 0 (- (length index) 1)))
                    (format t "~5t~a~25t~13,5g~40t(~,6g)~%"
                            (select nn i)
                            (select subbeta i)
                            (select subse i)
                            ))
            )
    )
  )

(defun formula-display
  (beta cov-mat model-formula &key (block-only t) (intercept t))
"Display a model summary for a model with coefficient estimates
beta and covariance estimates cov-mat and design matrix from
model-formula.  This should really be a method for the relevant
object. Little checking is performed: garbage in; garbage out"
(let* (
       (xintercept (if intercept 1 0))
       (nblocks (length (fourth model-formula)))
       (varnames (if intercept (cons "Intercept" (second model-formula))
                     (second model-formula)))
       (block-indices (+ xintercept (fourth model-formula)))
       )
  (format t "~a~20t~a~35t~a~%" "Block" "Wald Chisq" "p-value")
  (if (equal (length beta)
             (+ xintercept (second (array-dimensions (first model-formula)))))
      (dolist (i (iseq (- nblocks 1) 0))
              (block-test (elt block-indices i)
                          beta
                          cov-mat
                          :names varnames
                          :blockname (elt (third model-formula) i)
```

```
                          :block-only block-only))
    (error "length of beta doesn't match design matrix")
    )
  )
)

(defproto model-formula-proto
  '(formula design-matrix name-list block-names block-indices)
  '()
  *object*
  "Model formula")

(defmeth model-formula-proto :formula (&optional (new nil set))
"Set or return model formula"
  (when set
        (setf (slot-value 'formula) new)
        (send self :compute))
  (slot-value 'formula))

(defmeth model-formula-proto :isnew (&key formula)
  (send self :formula formula)
  )

(defmeth model-formula-proto :design-matrix (&optional (new nil set))
"Sets or returns design-matrix"
  (when set
        (setf (slot-value 'design-matrix) new))
  (slot-value 'design-matrix))

(defmeth model-formula-proto :name-list (&optional (new nil set) )
"Sets or returns list of column names"
  (when set
        (setf (slot-value 'name-list) new))
  (slot-value 'name-list))

(defmeth model-formula-proto :block-names (&optional (new nil set))
"Sets or returns list of block names"
  (when set
        (setf (slot-value 'block-names) new))
  (slot-value 'block-names))

(defmeth model-formula-proto :block-indices (&optional (new nil set) )
"Sets or returns list of column names"
  (when set
        (setf (slot-value 'block-indices) new))
  (slot-value 'block-indices))
```

```
(defmeth model-formula-proto :compute ()
"Internal use"
(let* (
       (result (formula (send self :formula)))
       )
  (send self :design-matrix (first result))
  (send self :name-list (second result))
  (send self :block-names (third result))
  (send self :block-indices (fourth result))
))

(defun as-formula (x) (send model-formula-proto :new :formula x))
```

5 Chunk index

References

Atkinson, E. N. (1995). Interactive dynamic graphics for exploratory survival analysis. *The American Statistician* **49**, 77–84.

Fleming, T. R. & Harrington, D. P. (1991). *Counting Processes and Survival Analysis*. Wiley: NY.

Grambsch, P. & Therneau, T. (1994). Proportional hazards diagnostics and tests based on weighted residuals. *Biometrika* **18**, 515–526.

Hakulinen, T. (1982). Cancer survival corrected for heterogeneity in patient withdrawal. *Biometrics* **38**, 933.

Harrell, F. & Lee, K. (1986). Verifying assumptions of the proportional hazards model. In *Proceedings of the Eleventh Annual SAS Users Group International*, pages 823–828.

Li, Q. H. & Lagakos, S. W. (1997). Use of the Wei–Lin–Weissfeld method for the analysis of a recurring and a terminating event. *Statistics in Medicine* **16**, 925–940.

Lin, D. Y. & Wei, L. J. (1989). The robust inference for the cox proportional hazards model. *Journal of the Americal Statistical Association* **84**, 1074–1078.

Ramsey, N. (1993). Literate-programming tools can be simple and extensible. Technical report, Department of Computer Science, Princeton University.

Therneau, T. & Hamilton, S. (1997). rhDNase as an example of recurrent event analysis. *Statistics in Medicine* **16**, 2029–2041.

Wei, L. J. & Glidden, D. V. (1997). An overview of statistical methods for multiple failure time data in clinical trials. *Statisics in Medicine* **16**, 833–839.

Wei, L. J., Lin, D. Y., & Weissfeld, L. (1989). Regression analysis of multivariate incomplete failure time data by modeling marginal distributions. *Journal of the Americal Statistical Association* **84**, 1065–1073.