APPENDIX AVAILABLE ON REQUEST

Research Report 140

Extended Follow-Up and Spatial Analysis of the American Cancer Society Study Linking Particulate Air Pollution and Mortality

Daniel Krewski et. al.

Appendix C. Computer Program for Random Effects Cox Model Using the Cox–Poisson Program

Note: Appendices Available on the Web appear in a different order than in the original Investigators' Report. HEI has not changed these documents.

Correspondence may be addressed to Dr. Daniel Krewski, McLaughlin Centre for Population Health Risk Assessment, Room 320, University of Ottawa, One Stewart Street, Ottawa, ON K1N 6N5, Canada.
E-mail: cphra@uottawa.ca.

Although this document was produced with partial funding by the United States Environmental Protection Agency under Assistance Award CR–83234701 to the Health Effects Institute, it has not been subjected to the Agency's peer and administrative review and therefore may not necessarily reflect the views of the Agency, and no official endorsement by it should be inferred. The contents of this document also have not been reviewed by private party institutions, including those that support the Health Effects Institute; therefore, it may not reflect the views or policies of these parties, and no endorsement by them should be inferred.

This document was reviewed by the HEI Health Review Committee
but did not undergo the HEI scientific editing and production process.

APPENDIX C: Computer Program for Random Effects Cox Model Using the Cox-Poisson Program

INTRODUCTION

General

The Cox-Poisson program is designed to estimate random-effects Cox proportional hazard survival models. It differs from the survival and GLM modules of general statistical systems such as SAS, R, S-Plus, and Stata in two main ways:

- It is designed to handle large data sets efficiently.
- Random effects are allowed to have a more complicated covariance structure than most other programs support.

There are two ways to use the program: as a stand-alone system, and through the "R" interface. The stand-alone version is somewhat restricted in what it allows: for example, all variables in the data file must be numeric. The R interface is much more flexible, allowing factors, interactions, transformations of variables, and generally most of what is allowed in the R language. Here we concentrate on using the R interface to the program, rather than the stand-alone version, which is now used mostly for development, troubleshooting, and debugging. It is assumed that the reader is at least somewhat familiar with the R language. For information on R, see http://lib. stat.cmu.edu/R/CRAN

The R system must be installed before the Cox-Poisson program. To install the latter, follow the instructions in the file "ReadMe.txt" which is supplied with the distribution package. This is an experimental program: there is no guarantee offered that it will be useful for any purpose.

The program currently can estimate proportional hazard survival regression models with random effects. Other capabilities are planned for the future.

Random Effects

Random effects are unobserved heterogeneities in "clusters", which might be regions or other groupings: a cluster is a subset of the study subjects, and the random effect associated with a given cluster i is a positive random variable U_i, whose expected value is taken to be 1. Clusters can be arranged in multi-level hierarchies, for example cities at the first level and postal codes at the second. Clusters at a particular level are disjoint, and each cluster at one level is a subset of a cluster (the *parent* cluster) at the next lower level. The random effects measure varying risk of clusters: a value larger than 1 indicates higher than typical risk to the members of the cluster, a value of 1 is neutral, and less than 1 indicates lower than typical risk. Random effects models differ in the assumed form of the covariance matrix of the random effects U. For a one-level system of clusters, the covariance matrix is defined as

$$D_{ij} = \mathrm{cov}(U_i , U_j)$$

In a multi-level cluster system, there is a covariance matrix for each level. The program supports several models for random effects covariance, which will be described later.

FUNCTIONS

The Cox-Poisson code is accessed from an R session, by using certain predefined functions for various purposes. Here we give brief summaries of the general functions, and in later sections the problem-specific functions will be described. Appendix C2 gives full descriptions of the functions and their arguments. These functions can be interspersed freely with other R statements and functions that may be useful for setting up the problem run. The next section gives an example of an R script for a survival estimation, which is reasonably typical of how the program works with R.

The first function that must be invoked in a session is CoxP ( ). This sets up the other functions, and needs to be executed only once in a session. It can be entered at the console at the start of the R session, or included as the first command in a problem script. Here is a list with brief descriptions of the other functions.

randEffects	Describes the random effects model to the program; there are arguments for specifying type of model, clustering variables, distance matrices, etc.
survProps	Groups the essential information for a survival model: end time variable, event indicator, and possibly some optional items.
describe.primary.data	If the data is to be read from a file, this gives characteristics of the primary data set. Unnecessary if the primary data is in a data frame.
Describe.secondary.data	A survival model can have a "secondary data set", described in the section on "Survival Problems", and this function defines the data set for the program. Unnecessary if the secondary data set is in a data frame.
describe.factor	If any covariates or other variables are "factors" in the R terminology, and if the primary data is in a file, then the factor's properties must be defined for the program. Unnecessary if the factor is in a data frame.

RobustVariance	Specifies to the program that the standard errors of regression coefficients are to be computed using a "robust" variance estimator, and gives some grouping information.
CoxPoiss	This is the function that does the actual estimation of a survival model. Its arguments accept definitions of random effects, data source, regression model, and other information, and it returns an R object of type "CoxPoiss" (or a list of such objects, in a multiple estimation run), which can be queried for estimation results.
summary.CoxPoiss	Prints on the screen, and optionally writes to a file, a summary description of the results contained in a "CoxPoiss" object.
RestoreCoxP	Brings back in an object of class "CoxPoiss", previously saved to disk, for further processing.
clustgroup	Groups two or more cluster-variable names together for multi-level random effects models: clustgroup (x, y) is equivalent to the standard R function c("x", "y"), and either can be used in defining the cluster hierarchy to the randEffects function.

AN INITIAL EXAMPLE

The "Rat data" is a well-known survival data set first published by Mantel and colleagues (1977). It describes a carcinogenicity experiment performed on 150 female rats, three from each of 50 litters. In the version we use, here are the first few records (the full data set is included with the Cox-Poisson package in the testing directory):

Indiv	**Litter**	**SurvTime**	**Event**	**Treat**
1	1	101	0	1
2	1	49	1	0
3	1	104	0	0

4	2	104	0	1
5	2	102	0	0
6	2	104	0	0
7	3	104	0	1
8	3	104	0	0
9	3	104	0	0
10	4	77	0	1

The survival time SurvTime is in weeks, and the variable Event indicates death; the variable Treat indicates that one rat in each litter was treated with the test substance. This is a small survival problem, and is easily handled by the survival code supplied with the R package (written by Terry Therneau of Mayo Clinic). First we show how to set up the rat problem for this code, then for the Cox-Poisson program. We define the survival model by the survival time and event indicator, and take Treat as a covariate in the regression. Litters are considered independent, but we expect rats within a litter to be correlated. Since litters may vary, we consider a random effect based on litter, with inter-litter correlations taken to be zero. Since the litters are not in a hierarchy but all on the same level, we take a one-level cluster structure, with litters as clusters. Here is an R script to estimate this model using the Therneau code:

```
# Go to working directory
setwd ("tests/femrat")

# Read data file into a data frame, and call it "ratframe"
ratframe <- read.table ("Femrat.dat", header = TRUE)

# Define the regression model
ratmod <- Surv(time=SurvTime, event=Event) ~ Treat + frailty(Litter)
```

```
# Estimate the model
ratobj <- coxph(formula = ratmod, data = ratframe, method = "breslow")

# Display the results
summary (ratobj )
```

The read.table function reads the data file, creates a data frame called "ratframe", and puts the data into it. The function Surv groups the survival time and event variables, just as does the function survProps in the Cox-Poisson program. The right side of the model definition states that the variable Treat is a regression covariate, and the function frailty tells the system to take the litters as defining random effects (sometimes called "frailties"). Finally, the last line displays the results of estimation on the screen. The display looks like this:

Call:

coxph(formula = ratmod, data = ratframe, method = "breslow")
n = 150

	coef	se(coef)	se2	Chisq	DF	p
Treat	0.906	0.323	0.319	7.88	1.0	0.005
frailty(Litter)				16.89	13.9	0.250
	exp(coef)	exp(-coef)	lower .95	upper .95		
Treat	2.47	0.404	1.31	4.65		

Iterations: 6 outer, 20 Newton-Raphson
Variance of random effect = 0.474 I-likelihood = -181.1
Degrees of freedom for terms = 1.0 13.9
Rsquare= 0.215 (max possible = 0.916)
Likelihood ratio test = 36.3 on 14.8 df, p=0.00145
Wald test = 7.88 on 14.8 df, p=0.924

This listing gives, for the sole covariate Treat, the regression coefficient, the standard error by the usual formula, the robust standard error (se2), some significance information, the relative risk

associated with Treat (exp (coef)), the variance of the random effect, and some more significance information.

Doing the same estimation with the Cox-Poisson program is very similar; here is an R script:

```
# Go to working directory
setwd("tests/femrat")

# Needed once only:
COXP ( )

# This name will appear on all output, along with a unique ID code
runname         <- "FemRat"

# names of output files:
logFilname      <- "FemRat.log"
outname         <- "FemRat.out"

# Title for output listings:
outhead         <- "Demonstration of Cox-Poisson Program with Rat Data"

# Read the data file into the data frame "ratframe"
ratframe <- read.table("FemRat.dat", header = TRUE)

# Define the model formula: no frailty term here
ratmod <- survProps(endtime = SurvTime, event = Event) ~ Treat

# Describe the random effects model (replaces frailty term)
reffs <- randEffects(clusters = Litter, type = "1LevelIndep")

# Do the estimation
ratobj <- CoxPoiss(model=ratmod, primary=ratframe, RandomEffects=reffs,
RunName=runname, logFile=logFilname)

# Display the results on the screen without the random effect values
summary (ratobj )

# To print to a file instead of the screen, with random effect values
summary(ratobj, printrandeff = TRUE, file = outname)
```

After defining a few names, this script reads the file into a data frame, and defines the regression model. The definition is nearly the same as for the Therneau code, but the function survProps is used instead of Surv, and there is no frailty term. Instead, the random effects model is defined in the next statement, which tells the program that the clusters are defined by the variable Litter, and the type of the model is: one level of clusters, with different clusters taken as independent. This is the only random effects model the Therneau code handles, so it doesn't have to be specified there, but here there are other models, and we must say which one we use. Some of the more complicated models require other information to be specified to the randEffects function, but "one-level independent" needs nothing else. When the random effects model has been defined, we are ready for the actual estimation process. The CoxPoiss function accepts the model name (called "model" here, instead of "formula"), and the data source (called "primary", because there can also be secondary data sources). We must also tell it the random effects definition, the run name, and the name of the log file, on which an account of the estimation process is written (useful mainly if something goes wrong). If any argument of CoxPoiss is not needed (e.g., if no log file is wanted, or there are no random effects in the problem), then just omit it. The results printed on the screen by the first summary command are:

```
---------------------------------------------------------------
Cox-Poisson program version 8.21
Run identifier: FemRat_25818
Estimation run Tue Sep 13 16:00:48 2005

CoxPoiss estimation with: No random effects
---------------------------------------------------------------
Model: survProps(endtime = SurvTime, event = Event) ~ Treat
Call:
CoxPoiss(model = ratmod, primary = ratframe, RunName = runname,
    RandomEffects = reffs, logFile = logFilname)

Sample Size: 3533
```

```
Final Log-Likelihood: -208.845

          Coefficient     Std. Error         t         exp(Coef)      Lower 95%      Upper 95%
Treat     0.898225     0.218218        4.11618      2.45524         1.60084          3.76566

Wald statistic for H0:{all coeffs = 0} = 16.942978 on 1 d.o.f.
-------------------------------------------------------------------
Cox-Poisson program version 8.21
Run identifier: FemRat_25818
Estimation run Tue Sep 13 16:00:48 2005

CoxPoiss estimation with: One-Level, Clusters Independent
-------------------------------------------------------------------
Model: survProps(endtime = SurvTime, event = Event) ~ Treat
Call:
CoxPoiss(model = ratmod, primary = ratframe, RunName = runname,
     RandomEffects = reffs, logFile = logFilname)

Sample Size: 3533
Final Log-Likelihood: -198.538

          Coefficient       Std. Error      t          exp(Coef)     Lower 95%     Upper 95%
Treat        0.902085       0.227778     3.96036      2.46474         1.57719         3.85173

Wald statistic for H0:{all coeffs = 0} = 15.684455 on 1 d.o.f.

Random effects dispersion parameters:
SigmaSq
0.295547
```

This gives the result of two estimations: if random effects are specified, the program always begins by estimating the survival model with no random effects, and the result is the first listing of coefficients above. Although it is not given here, when the no-random-effects model is estimated with the Therneau code, the results are identical to those listed for the Cox-Poisson code. The second listing gives the results with the specified random effects model. The name SigmaSq refers to the variance of the random effects. If requested, the summary function will also list the values of the random effects for all clusters.

Comparing the results of the two codes is a little complicated, because the random effects in the Therneau code correspond roughly to the logarithms of the Cox-Poisson random effects. Also, the two codes use different distributional assumptions about the random effects, and different estimation methods. Taking these differences into account, the two codes give approximately similar results.

SURVIVAL PROBLEMS

The general form of an R script for estimating a survival model is shown by the rat-data script in the last section. There are statements defining auxiliary information such as the run heading, run name, etc., statements that read the data or define the properties of the data file, possibly statements that modify the data or define new variables, statements that define the regression model, the random effects, and other structures the program uses, and finally the CoxPoiss statement that carries out the estimation. Then there are statements that extract the results from the object returned by CoxPoiss.

The main arguments of the CoxPoiss function are described below: several of them are illustrated in the last section with the rat data. Arguments have the form keyword = value; the keyword is fixed and the value is supplied by the user. The important keywords are listed below. The only arguments that are not optional are model and primary, both of which are illustrated above with the rat data. All other arguments have default values, and need be present only if the defaults are not appropriate. Appendix C2 describes all arguments of CoxPoiss, along with those

of the other functions listed in section “Functions”. Here we just describe the most important and frequently used arguments.

model (no default): an R model formula (or the name of a previously defined formula, as in the rat-data example). This has the form response ~ expression. The “response” side of the formula, the part before the “~”, is the function survProps and its arguments, which identify the end time and event-indicator variables, and optionally a start time and/or time origin variable(s). The right hand side, after the “~”, is any expression in covariates that the R formula syntax allows. Specifically, transformations such as log (x) are allowed, along with interactions such as x* z, and terms are separated by “+”. To indicate an arithmetic operation, enclose it in I ( ): thus I (x + z) means the variable which is the sum of the variables x and z, while x + z means “use the two variables x and z as covariates”. There can also be an offset term, which has the form offset (x) and which specifies that the variable x is to be a covariate whose coefficient is fixed at 1. For a description of the R formula notation, see the manual “Introduction to R” (supplied with the R package), section “Statistical Models in R”. One exception to the rule that any R-valid formula is allowed is this: a covariate that is in a secondary data file may not be transformed, or enter into expressions or interactions; so expressions like log (x) are legal only if x is in the primary data file. An example of a formula definition is

```
mform <- survProps(endtime = survtime, event = death) ~
  curcig + edulow + bmi + alc + log(pm25) + offset(rd)
```

primary (no default): The primary data set is the main source of data for the run. Secondary data sets are discussed below: most runs will not have a secondary set. The specification

of the primary data set can be either the name of a data frame, or the name of a "DataSource" object as returned by the describe.primary.data function. If it is possible, the simplest way to get data to the program is just what is done in the rat-data example: read it into a data frame, and then give the data frame's name in the primary argument. This will be possible unless the data set is too large to fit in a data frame. For that situation, we need the describe.primary.data function, which is discussed in the section "Large Data Files" below.

RunName (default "CoxPoissRun"): This is any string, containing no embedded blanks; it is the user's choice of a name for the present run. In the output, a 5-digit random number will be appended to it for additional uniqueness. The run name can be useful for establishing an audit trail in a study.

outheading (default NULL): This is any string, and is used as a title in output.

RandomEffects (default NULL): As in the rat-data example, a random effects specification is created and named using the randEffects function, and the name is then given to the CoxPoiss function with the RandomEffects argument. If the problem does not involve random effects, just omit this argument. More complicated random effects specifications are discussed in the section "Random Effects Models" below.

strata (default NULL): The data records can be partitioned into disjoint subsets called strata; each stratum is postulated to have its own baseline hazard function, but regression coefficients and random effects are estimated for all records and assumed to apply to all strata. Stratification is specified by a variable: the argument strata = Sex means that Sex is a variable in the data set, and each distinct value (presumably two of them, in this case)

of Sex defines a stratum. The stratum variable can have any number of values, and can be either numeric or a factor.

logFile (default NULL): A log file records details of the estimation process, and can be a useful aid to troubleshooting. If a log file is desired, give a filename with all needed path information. Filenames should not contain embedded blanks.

robustVarSpec (default NULL): Ordinarily, different data records are assumed independent. However, in some cases the data might have been "cluster-sampled" in some way. The most obvious example is that of repeated events (e.g. hospitalizations, spells on welfare, etc.) for each subject: we would expect that different observations (data records) for the same subject would be correlated. In this case, the standard errors determined by the usual procedure are wrong, and we need a variance calculation that is robust to the cluster-sampling. Suppose there is a variable in the data, group, say, such that each value of group represents a group of possibly correlated observations, but that observations with different values of group are independent. Then we can define a specification

```
robvar <- RobustVariance(groupVarname = group, UseRobust = FALSE,
      ShowRobust = FALSE)
```

and include the argument robustVarSpec = robvar.

maxiterations (default 50): The maximum number of iterations to be used in the estimation process. Must be a positive integer. If the program reaches the maximum number of iterations with no convergence, it reports the current unconverged values in the output file, so when trying a new model it is a good idea to check the log file to see if the process converged. If not, try increasing the iteration limit.

tolerance (default 1.0e-6): The convergence tolerance, applied to both the estimated regression coefficients and the random effects dispersion parameters.

secondary (default NULL): A secondary data set is specified to the CoxPoiss function by secondary = secspec, where secspec is either the name of a data frame, or the name of an object returned by the function describe.secondary.data. Since secondary data sets will rarely if ever be too big to fit in a data frame, the normal (and simpler) case will be to read the secondary file into a data frame, say df, with read.table, then give the data frame's name to CoxPoiss with secondary = df. The use of secondary tables is described below in the section "Time-Dependent Covariates and Secondary Data Tables".

RANDOM EFFECTS MODELS

General

Random effects are associated with "clusters", which are disjoint groups of subjects (operationally, of input records). In a two-level model each cluster is partitioned into subclusters, and there may be further levels of subdivision. As an example, the clusters might be the cities of residence of individuals in the sample, and subclusters might be postal codes. Each city i will have its level-1 random effect U_i, and each postal code p in city i will have a level-2 random effect u_{ip}.

Clusters are defined to the program by cluster variables: to continue the city-postal code example, the data set would have a variable City, say, and a variable postcode, say, and the pair

of values for these two variables uniquely defines each data record's place in the cluster hierarchy.

Covariance models are assumptions about the form of the covariance matrix $\{\mathrm{Cov}(U_i, U_j)\}$ of the level-1 random effects, and the form of the conditional covariance (conditioned on U_i) of the level-2 random effects $\{\mathrm{Cov}(u_{ip}, u_{iq} \mid U_i)\}$, for all i, p, q. Each covariance model will have certain numeric parameters, called dispersion parameters, that must be estimated from the data. The program currently implements four types of covariance model for survival problems.

Covariance Model Types

Here is a list of the covariance models the program supports for survival problems. A random effects specification is made with the randEffects function, which is described below. One of the arguments to that function is type, and the following list gives the allowed values for type.

1LevelIndep

For the 1LevelIndep model, there is only one level of clustering. The covariance matrix is assumed to be

$$\mathrm{Cov}(U_i, U_j) = \sigma^2 \mathbf{I}$$

where $\mathbf{I}$ is the identity matrix, and σ^2 is a dispersion parameter to be estimated, called "SigmaSq" in the program's output. The only information that has to be specified to the

program is the clustering variable, given in the rat example above as "clusters = Litter". Since this is a one-level model, only one clustering variable is supplied.

1LevelDistDecay

The distance-decay models assume that the covariance between the random effects at a particular level has the form (in full generality)

$$\mathrm{Cov}(U_i, U_j) = \sigma^2 \, w_i \, w_j \, \rho^{2d(i,j)}$$

where $\sigma > 0$ and $\rho \geq 0$ are dispersion parameters to be estimated, and $d(i,j)$ is an inter-cluster distance matrix that must be supplied as data. The expressions w_i and w_j are values of a known cluster-weight vector that can optionally be supplied as data; this allows specification of "gravity" type models of covariance; if the weight specification is omitted, the weights are taken to be 1. The value of ρ must usually be restricted to be less than some bound, in order to insure that the distance matrix will be positive definite; this bound is calculated by the program. An example of a 1LevelDistDecay specification is

```
dmdf <- read.table("L1DistMatNm156cit1.dat", header=TRUE)

reffs <- randEffects(clusters = city, type = "1LevelDistDecay",
DistMatLev1 = dmdf)
```

This illustrates the point that distance matrices can be (and usually are) supplied in a data frame. This specification omits a weight vector. The form of a distance matrix is described in Appendix C1, which also describes the DistScalFactor option (for conversion of distance units).

2LevelDistDecay

For 2LevelDistDecay, we need to specify the inter-cluster distance matrix, but also, for each cluster, a distance matrix for its subclusters must be supplied. All these subcluster matrices are given in one file, as explained in Appendix C1. We can also specify cluster weights, as for the 1-level model, but we can give weights for either level.

The level-1 (inter-cluster) covariance matrix (supposing no weights) is assumed to be of the form

$$\text{Cov}(U_i, U_j) = \sigma_1^2 \rho^{d_1(i,j)}$$

with all symbols having the same meaning as in the 1-level distance-decay model. For each cluster i, the level-2 covariance matrix for the subclusters of cluster i, conditional on the value U_i, is

$$\text{Cov}(U_{ip}, U_{iq} \mid U_i) = \sigma_2^2 \rho^{d_{2i}(p,q)}$$

where $d_{2i}(p, q)$ is the distance matrix for the subclusters of cluster i.

For a two-level problem, there are two clustering variables, specified as a list, and an example of a two-level control specification is

```
dmdf1 <- read.table("L1DistMatNm156cit1.dat", header = TRUE)

dmdf2 <- read.table("L2DistMat155cit.dat", header = TRUE)
reffs <- randEffects(clusters = clustgroup(city, postcode),
type = "2LevelDistDecay", DistMatLev1 = dmdf1, DistMatLev2 = dmdf2)
# Note: clusters = c("city", "postcode") also works
```

Another option for 2LevelDistDecay is: we can specify $\rho = 0$ at either or both levels. This is done by including FixRhoValueLev1 = 0, or FixRhoValueLev2 = 0. You can actually specify any value, not just zero, but zero would be by far the most common case. Specifying $\rho = 0$ at either level is equivalent to specifying an independent-clusters model at that level, i.e. the conditional covariance matrix is diagonal. There is then no need to give a distance matrix for that level, and the distance matrix specification is ignored if it is supplied. As an example, suppose we think that cities are independent, but within a city, zipcode-based effects are assumed to follow a distance-decay model with a contiguity matrix. The specification would be:

```
reffs <- randEffects(clusters = clustgroup(city, postcode),
type = "2LevelDistDecay", FixRhoValueLev1 = 0, DistMatLev2 = dmdf2)
```

Here we could have also given a level-1 distance matrix filename, but because of the FixRhoValueLev1 = 0 specification, it would be ignored. It is not considered an error, since one might want to quickly test a $\rho = 0$ specification without major changes to the control file. To specify $\rho = 0$ at the subcluster level, use FixRhoValueLev2 = 0. Specifying $\rho = 0$ at both levels is equivalent to a 2-level independent-clusters model.

There is one other specification for the 2-level model: at the subcluster level, we can have one σ^2 and ρ pair that is assumed to specify the subcluster covariance matrices within all clusters, or we can have (σ_1^2, ρ_1), (σ_2^2, ρ_2), (σ_3^2, ρ_3), . . . , (σ_m^2, ρ_m), individually for each cluster. The default is a single (σ^2, ρ) value for all clusters. To specify either, we include SubclustersSimilar = FALSE or SubclustersSimilar = TRUE (default).

MultiLevelIndep

Here the cluster tree has any number of levels, and the *U*'s at any level are assumed conditionally independent, given the *U*'s at the parent level. The first thing needed to specify this model is the list of cluster-variables; these define the cluster hierarchy. The specification might look like this for a three-level model:

reffs <- randEffects(clusters = c("state", "city", "zipcode"), type = MultiLevelIndep)

The only other specification accepted by this model is the same SubclustersSimilar parameter as described above for 2-level models. The default is SubclustersSimilar = TRUE.

Function randEffects

All information about a particular random effects model is specified to the program by the randEffects function, which appears in a simple form in the rat-data example above. The procedure is: use the r andEffects function to create a named random effects object with the desired properties, then give its name as the value of the RandomEffects argument of CoxPoiss. The arguments of the randEffects function give the information that defines the particular model. Here are the main arguments for the randEffects function, and their values:

Keyword	**Acceptable Values**
Clu s te r s	One or more variable names (see below)
Type	For survival models, one of "1LevelIndep", "2LevelIndep", "1LevelDistDecay", "2LevelDistDecay", or "MultiLevelIndep".

FixRhoValueLev1	Numeric value to fix level 1 “rho” parameter in distance decay models; usually 0 if used.
FixRhoValueLev2	Numeric value to fix level 2 “rho” parameter; usually 0 if used.
DistMatLev1	Data frame name, or filename and path, if needed, of level 1 distance matrix, in distance decay models. For distance matrix formats see Appendix C1
DistMatLev2	Data frame name, or filename and path, if needed, of level 2 distance matrix.
DistScaleFactorLev1	Numeric scale factor for level 1 distance matrix (e.g. to convert distance units). Default 1.
DistScaleFactorLev2	Numeric scale factor for level 2 distance matrix. Default 1.
ClustWeightVecLev1	Level 1 weight vector: existing R vector, or file name and path. Default NULL.
ClustWeightVecLev2	Level 2 weight vector: existing R vector, or file name and path. Default NULL.
SubclustersSimilar	In 2-level or multi-level model, should clusters all have same subcluster parameters? Values: one of TRUE, FALSE. Default TRUE.

The interpretation of these is just the same as in the stand-alone program. Any argument that does not apply can be omitted. The DistMatLev1 and DistMatLev2 arguments can specify a data frame or connection instead of a filename.

The “Clusters” argument needs mention: a one-level model requires one variable to define the clusters, e.g. City, and a *k*-level model requires *k* variables, e.g. City, postcode for two levels. There are two ways to give the names:

- We can use the standard R vector-definition function c( ) to define a character vector:

  Clusters = c(“City”, “postcode”)

  Note the presence of quotation marks.

- We can use the special function clustgroup:

  Clusters = clustgroup(City, postcode)

  Note the absence of quotation marks.

Either of these can be used, at the convenience of the user. Cluster variables are specified in order of level, i.e. the specification abovc says that City defines the level-1 clusters, and postcode the level-2 subclusters. For one level, only one cluster variable is needed, and clustgroup ( ) or c( ) can be omitted, as in the rat-data example clusters = Litter above. Each unique value of the cluster variable corresponds to one cluster. The cluster variable can be numeric (usually integer), or a factor, such as city names. There is no need for integer values to be consecutive: any set of numbers will do, such as U.S. postal codes (“zipcodes”), which are 5-digit numbers.

LARGE DATA FILES

The easiest way to handle any problem whose main data set is small enough, is to read the data into an R data frame, as illustrated previously by several examples. The Cox-Poisson program

stores data more compactly than R, so that it can handle larger data sets than a data frame can hold, but it is restrictive in the data it accepts directly. We can combine the flexibility and power of R, with the capacity of Cox-Poisson, by using R to translate a large file, a section at a time, to the form needed by Cox-Poisson. In translating, it makes transformations such as the logarithm (as in an example above), expands factors and interactions, etc., and writes out the results in a binary file that the Cox-Poisson program can read. The data file can contain many more variables than are used in any one run: only those actually used will be written to the binary file, so the extra ones do not take up space. When the binary file and control information is written, the Cox-Poisson program is invoked automatically, and the results are then returned to R. The translation process requires some control by the user, which we describe now.

To translate correctly a file which must be processed in sections, the program must have some information about the file's contents before starting, specifically the names and types of the variables in the file. The user provides this information with the describe . primary.datafunction. The file "ACSlike3.dat", supplied with the Cox-Poisson in the test directory, is a test file designed to look like the ACS data set, but with fictional data values. The cities are represented by numeric codes. Here is how we describe it for translation:

```
ACSNames <- c("id","city","zipcode","subclid","strata","failtime", "censor", "curcig",
   "evpconly", "smkcpd", "xsmkcpd", "smkcyr", "xsmkcyr", "passive", "edulow",
   "indusexp", "bmi", "alc", "fine", "dummy")

ACSTypes <- list("integer", "integer", "integer", "integer", "integer", "real", "integer",
   "integer", "integer", "integer", "integer", "integer", "integer", "real", "integer",
   "integer", "real", "real", "real", "integer")

datdescr <- describe.primary.data(dataSource = "ACSlike3.dat", varnames = ACSNames,
   vartypes = ACSTypes)
```

Instead of “integer” and “real”, we could use “1” and “2”, respectively. We could also leave out the varnames and vartypes arguments, and supply the names and types (in the “1-2” code) as the first two lines (aside from comment lines) of the data file.

This works as long as all the variables in the file are numeric; but we also want to be able to use factors. A factor might have character values, such as “Male” and “Female”, or numeric codes as values. Either way, we must describe them to the program. The file “ACSlike3CNames.dat” is like the file “ACSlike3.dat”, except that the cities are denoted by six-letter codes, with the first two letters a state-name abbreviation (“AL” for Alabama, “CA” for California, etc.), and the last four a city-name abbreviation, so e.g. “ALBIRM” is Birmingham, Alabama. The variable “city” is a factor with these city-codes as values (47 cities in all). Also, the variable “edulow” is considered a factor with the numeric codes 0, 1, 2, representing “Yes”, “No”, and “Unknown”. We would like to have these descriptions in the output instead of the numeric codes. We use the function describe.factor as follows:

```
ACSNames <- c(“id”, “city”, “zipcode”, “subclid”, “strata”, “failtime”, “censor”, “curcig”,
    “evpconly”, “smkcpd”, “xsmkcpd”, “smkcyr”, “xsmkcyr”, “passive”, “edulow”,
    “indusexp”, “bmi”, “alc”, “fine”, “dummy”)

CityNames <- c(“ALBIRM”, “ALHUNT”, “ALMOBI”, “ARLROC”, “AZPHOE”,
    “AZTUCS”, “CAANAH”, “CAFRES”, “CALANG”, “CARIVE”, “CASACR”,
    “CASBAR”, “CASDIE”, “CASJOS”, “COCOLO”, “CODENV”, “COFCOL”,
    “COGREE”, “COPUEB”, “CTBRID”, “CTHART”, “CTNHAV”, “DCWASH”,
    “DEWILM”, “FLFLAU”, “FLORLA”, “FLTAMP”, “GAATLA”, “GACOLU”,
    “GASAVA”, “IDBCIT”, “ILCHIC”, “INEVAN”, “INGARY”, “ININDI”, “INSBEN”,
    “INTHAU”, “IOCRAP”, “IODMOI”, “IODUBU”, “IOWATE”, “KSTOPE”,
    “KSWICH”, “KYLEXI”, “LABATO”, “LANORL”, “LASHRE”)

Citfact <- describe.factor(levelvals = CityNames)

Edufact <- describe.factor(levelvals = c(0,1,2), levelnames = c(“Yes”, “No”, “Unknown”))
```

```
ACSTypes <- list("integer", Citfact, "integer", "integer", "integer", "real", "integer",
    "integer", "integer", "integer", "integer", "integer", "integer", "real", Edufact,
    "integer", "real", "real", "real", "integer")

datdescr <- describe.primary.data(dataSource = "ACSlike3Names.dat", varnames =
    ACSNames, vartypes = ACSTypes)
```

In describing a factor with describe.factor, we give all the values that it has in the file, in the argument levelvals. We can also rename these values, for use in the output listings, by the argument levelnames: the definition of Edufact above tells the program to look in the file for the values (0,1,2), but to replace them in the output by ("Yes", "No", "Unknown"). It is somewhat inconvenient to have to give all the possible values for a factor, but it is necessary. The simplest way to do it, if the list is long, is to keep the names in a file, and use read.table to read them into a character vector. The list of factor values specified with levelvals can include more names than are actually in the file (extras will be ignored), but cannot omit any that are in the file.

Finally, having described the data source, we specify its description to CoxPoiss:

```
CPobj <- CoxPoiss(model = modelACS, primary=datdescr, . . . )
```

It is not, of course, necessary to give a name to the data-source description: we could just enter:

```
CPobj <- CoxPoiss(model = modelACS, primary = describe.primary.data(dataSource =
"ACSlike3Names.dat", varnames = ACSNames, vartypes = ACSTypes), . . . )
```

The same is true of model, and of all the complex arguments. Generally, however, it is neater to define and name the complex arguments in advance, and use their names in CoxPoiss.

TIME DEPENDENT COVARIATES AND SECONDARY DATA TABLES

All time-dependent covariates must be piecewise-constant in time. There are two ways to handle time-dependent covariates: record repetition, and a secondary file. Record repetition, the method used by the Therneau survival code supplied with R, is best illustrated with an example: suppose a data set has covariates X, R, Z, and W: X and Z depend on both time and subject, W depends on subject but not on time, and R depends on time but not subject (although it might depend on a larger grouping such as city). Then we set up the data set so that each data record is repeated for each value of the TD covariates:

Id	**Start**	**End**	**Death**	**X**	**R**	**Z**	**W**	**…**
37	0	10	0	3.2	9.8	0	18.8	…
37	10	20	0	4.3	7.3	1	18.8	…
37	20	30	0	6.7	6.7	3	18.8	…
…	…	…	…	…	…	…	…	…
37	50	57	1	7.4	9.8	2	18.8	…
38	0	10	0	1.6	7.3	1	22.3	…
38	10	20	0	0.9	6.7	0	22.3	…
38	20	30	1	2.3	5.6	2	22.3	…
…	…	…	…	…	…	…	…	…

Here the records are shown for two subjects, 37 and 38. Each record has a start time and an end time, and the covariate values are those appropriate to the subject and the time interval. Since W is not time-dependent, its values for a subject are just repeated on each of the subject's records. The vaues of R are the same for both subjects (supposing that they live in the same city, for example), and the values of X and Z change with both time and subject. Different covariates do

not have to be “synchronized”, in the sense of changing values at the same time, but it is more efficient if they are. In any case, a new record is needed for every change in the value of any covariate.

If only one or a few covariates are time-dependent (TD), it is tedious and wasteful of memory to duplicate all the non-TD covariates many times just to accommodate the few TD covariates. An alternative that can save memory space, simplify data preparation, and in some cases save running time, is to use a secondary table. This is another data file, which lists the values of the TD covariates and corresponding time-intervals. It must also have a “key” variable, which determines the correspondence between the primary and secondary tables. If the individual’s ID is used as the key variable, then each record in the secondary file has an ID, a start time, an end time, possibly a time origin, and a value for each of the TD covariates for the specified time interval. An example of a secondary file is:

ID	**StTime**	**EndTime**	**Weight**	**Cholesterol**	**# Name-record**
307	0	5	76.3	38.2	# 1 st record for indiv 307
307	5	8	82	39.3	
307	8	12	84	39.7	
523	0	7	66.2	28.4	# 1 st record for indiv 523
523	7	15	69.1	29.3	

....etc .....

Here there are two TD variables, Weight and Cholesterol, given on adjacent time-intervals for each individual. There can be as many records for an individual as necessary. In this example,

the variable names and types are given in the data file, but they can also be given in the R script. The individual's ID serves here as the key variable, indicating that the records in the secondary file with a given value of ID correspond to the records in the primary file with the same ID value. The key variable must have the same name in both primary and secondary files.

In some cases, it may be more appropriate to use a different key variable. For example, a single air pollution monitor located in a city will be associated with every individual living in the city, and if the data give the time-dependent readouts for monitors located in several cities, then these variables will be indexed by city rather than by individual. The ACS data is structured like that. Here is the beginning of the fictional secondary file ACSlikeSec.dat, which is included in the test problems supplied with the package:

CITY	**STTIME**	**ENDTIME**	**PM10**	**# Name-record**
1	0.00	7.50	0.4900	# start of data for city 1
1	7.50	15.00	0.5710	
1	15.00	22.50	0.3373	
1	22.50	30.00	0.8295	
1	30.00	37.50	0.7811	
1	37.50	45.00	0.8075	
1	45.00	52.50	0.6050	
1	52.50	60.00	0.9405	
1	60.00	67.50	0.9059	
1	67.50	75.00	0.4687	

1	75.00	82.50	0.7135	
1	82.50	90.00	0.8204	
1	90.00	97.50	0.5377	
2	0.00	7.50	0.9249	# start of data for city 2
2	7.50	15.00	0.5341	
2	15.00	22.50	0.8151	
2	22.50	30.00	0.7458	
2	30.00	37.50	0.8822	
2	37.50	45.00	0.7415	
2	45.00	52.50	0.5872	
2	52.50	60.00	0.4977	
2	60.00	67.50	0.7124	
2	67.50	75.00	0.9238	
2	75.00	82.50	1.0184	
2	82.50	90.00	0.7842	
2	90.00	97.50	0.0482	
3	0.00	7.50	0.4544	# start of data for city 3
3	7.50	15.00	0.5999	
3	15.00	22.50	0.7713	
3	22.50	30.00	0.4948	
3	30.00	37.50	0.7268	
3	37.50	45.00	0.8765	

. . . . etc. . . . . .

The variable CITY corresponds to the same variable in the primary file, allowing the association to be made for each individual's city. Note: in this example we are representing cities by a numeric code instead of alphabetic names or codes, and the primary file must use the same numeric code. When, as in this case, the key variable has many fewer values than the number of individuals, the use of a secondary table can give a considerable saving in memory space, as well as some saving in running time.

In the file above, the set of time-intervals for each city are the same; this is not necessary: the records for each city can divide up time in any way desired (although two intervals for the same city can not overlap). However, if it is feasible, it is more efficient to make the intervals the same for all cities as above.

The simplest way to handle a secondary table is to put it into a data frame first. Normally secondary tables are small enough to fit into a data frame, and this is the easiest way to input factors. If we have created a data frame (using read.table or otherwise) called SecTab, containing the variables CITY STTIME ENDTIME PM10, with CITY a factor corresponding to the same variable in the primary file, then we can specify the secondary table to the system using the describe.secondary.data function:

```
secdat <- describe.secondary.data(dataSource = SecTab, keyvarname = "CITY", endtime =
"ENDTIME", starttime = "STTIME")
```

The only information we need to give is the data source, in this case the data frame SecTab, and the start-time, end-time, and key variables. If instead of a data frame, we specified a file name as the data source, then we would also have to specify the varnames and vartypes arguments, which work the same way as in the describe.primary.data function: in particular, we would have to use describe.factor to define the type of the variable CITY:

SecTypes <- list(describe.factor(levelvals = CityNames), "real", "real", "real")

secdat <- describe.secondary.data(dataSource="SecFile.dat", varnames = c("CITY", "STTIME","ENDTIME", "PM_{10}"), vartypes = SecTypes, keyvarname = "city", endtime = "endtime", starttime = "sttime")

To use the secondary table data, we would just specify the variable PM_{10} as a covariate like any other. There must not, of course, be a variable named PM_{10} in the primary file. Some of the test problems supplied with the program illustrate the use of secondary tables.

For a dataset in which most or all of the covariates are time-dependent, there is no advantage to using a secondary table, and the extra overhead it requires is a disadvantage.

REFERENCES

Cressie NAC. 1991. *Statistics for Spatial Data*. Wiley and Sons, New York, NY.

Ma R, Krewski D, Burnett RT. 2003. Random effects Cox models: A Poisson modelling approach. *Biometrika* 90:157-169.

Mantel N, Bohidar NR, Ciminera JL. 1977. Mantel-Haenszel analyses of litter-matched time-to-response data, with modifications for recovery of interlitter information**.** *Cancer Res* 37:3863-3868.

Therneau TM, Grambsch PM. 2000. *Modelling Survival Data*. Springer-Verlag, New York, NY.

Venables WN, Ripley BD. 2002. Generalized linear models. In *Modern Applied Statistics with S*, Springer Verlag, New York, NY.

Appendix C1 DISTANCE MATRIX FORMAT

The distance-decay covariance models require distance matrices to be supplied, that give (possibly artificial) distances between clusters or subclusters. The stand-alone program has somewhat restrictive requirements for the format of distance-matrix files. With the R interface, the rules are more flexible; consider the specification of, say, a one-level distance-decay model, for example:

ReffDD1 <- randEffects(clusters = City, type = "1LevelDistDecay", DistMatLev1 = "distmat47CityNames.dat", DistScaleFactorLev1 = 1000)

Here the DistMatLev1 argument specifies a filename; the first few lines of the file are:

```
ALHUNT ALBIRM 132.5
ALMOBI ALBIRM 340.5
ARLROC ALBIRM 528.3
AZPHOE ALBIRM 2337.4
AZTUCS ALBIRM 2251.4
CAANAH ALBIRM 2865.2
CAFRES ALBIRM 3006.7
CALANG ALBIRM 2910.1
CARIVE ALBIRM 2820.2
CASACR ALBIRM 3147.8
CASBAR ALBIRM 3024.8
CASDIE ALBIRM 2815.7
CASJOS ALBIRM 3244.2
COCOLO ALBIRM 3185.9
CODENV ALBIRM 1713.8
```

Each line gives the distance between the two named cities: since distances are symmetric, only one pair needs to be given (i.e. if ARLROC ALBIRM is given, then ALBIRM ARLROC need not be), and any pair of cities not in the list will be deemed to be at infinite distance (i.e. zero covariance). The distance matrix can be bigger than necessary, in the sense of containing cities

that are not in the data of the problem: any line with an unknown name is ignored. This facilitates running problems with subsets of the full data set: the same distance matrix can be used, without having to reduce it to only the clusters that are in the current run. The cluster names in the distance file should correspond to the values of the variable given in the clusters argument, in this example, City. If numeric codes are used instead of names, the same codes should be used in the distance matrix file. The diagonal entries of a distance matrix are always zero, since everything lies at zero distance to itself. No other zero values are allowed, i.e. any two distinct clusters must be at a positive distance.

One special type of distance matrix is the adjacency or nearest-neighbor type, in which all distances are either 1 or infinite. Any two clusters are either neighbors or total strangers; there are no degrees of neighborliness. There is no special notation for this type of matrix: distance values that are supplied are all given as 1, and omitted pairs have infinite distance.

The DistMatLev1 argument does not have to specify a filename; it can also be a "connection" (an open data stream), or a data frame (i.e. the distance-matrix file can be read into a data frame in advance; usually the simplest alternative), or the name of an existing matrix in the R workspace; in the last case, the row and column names of the matrix should correspond to the cluster names of the clusters variable. Again, it can have extra names not in the current problem's list of clusters. Using an explicit matrix is restricted to problems with relatively few clusters, since full matrices can be bulky: the form whose first few lines are given above can be regarded as a sparse-matrix notation.

The parameter DistScaleFactorLev1 is a scale factor: all distances will be divided by this value. Its default value is 1. This allows scaling of distances to put them into other units, or to give them reasonable ranges: for example, suppose that distances in the file are given in kilometers. The scale factor 1000 above converts to distances in 1000's of kilometers (i.e. in megameters). For North America, this puts all distances on a numeric scale of about 0 to 8. Recall that the covariances are assumed proportional to ^{dist,} where $0 < < 1$. An infinite distance implies the value 0. Even if is close to 1, a large distance (say, 5000) is effectively infinite: e.g. $(0.99)^{3000} = 8.05 \times 10^{14}$. This is why it's a good idea to scale the distance to a moderate range. For an adjacency-type matrix (all distances either 1 or infinite), the scale factor should be omitted.

A 2-level distance matrix works the same way, but specifying a level-2 cluster requires two variables. If the level-2 clusters are defined by City and Zipcode, then the distance matrix file might include

ALMOBI	23471	ALMOBI	23456	0.4476
ALMOBI	23471	ALMOBI	23854	1.1787
ALMOBI	23316	ALMOBI	23013	1.8433
ALMOBI	23471	ALMOBI	76013	2.9425
ARLROC	53471	ARLROC	56013	0.1919
ARLROC	53712	ARLROC	56013	0.2617
ARLROC	53741	ARLROC	57316	1.3207
ARLROC	53712	ARLROC	57316	0.7741
AZPHOE	86333	AZPHOE	87456	1.017
AZPHOE	87316	AZPHOE	87456	1.3313
AZPHOE	86643	AZPHOE	83982	0.3603
AZPHOE	83134	AZPHOE	83982	0.8062
AZTUCS	87316	AZTUCS	86333	1.0686
AZTUCS	86643	AZTUCS	86333	2.784

For a “neighbor” or “contiguity” type of distance matrix, the fifth column should be all 1’s. As with level-1 matrices, symmetric pairs need only one representative, and any omitted distance is assumed infinite. Also, a data frame, file name, or connection can be specified.

At present, a level-2 distance matrix must be strictly block-diagonal, i.e. subclusters can only be connected to other subclusters of the same cluster, as in the example above, where zipcodes within a city can be neighbors, but not zipcodes in different cities. Later versions of the program may relax this rule.

Appendix C2 FUNCTION REFERENCE

The functions CoxPoiss, and randEffects have already been described in earlier sections. We repeat the argument listings here for completeness. For each function, we give its definition header, a list of arguments with explanations, and a description of the value the function returns.

randEffects

Describes the the random effects model to the program. Definition and defaults:

```
randEffects <- function(clusters,
        type = c("1LevelIndep", "2LevelIndep", "1LevelDistDecay",
          "2LevelDistDecay", "MultiLevelIndep",
        FixRhoValueLev1 = NULL, FixRhoValueLev2 = NULL,
        RhoUpBndLev1 = NULL, RhoUpBndLev2 = NULL,
        DistMatLev1 = NULL, DistMatLev2 = NULL,
        DistMatTypeLev1 = c("NUMERIC", "CONTIG", "LATLONG"),
        DistMatTypeLev2 = c("NUMERIC", "CONTIG", "LATLONG"),
        DistScaleFactorLev1 = NULL, DistScaleFactorLev2 = NULL,
        ClustWeightFileLev1 = NULL, ClustWeightFileLev2 = NULL,
        SubclustersSimilar = TRUE
```

Arguments

Clusters: a one-level model requires one variable to define the clusters, e.g. City, and a k-level model requires k variables, e.g. City, postcode for two levels. There are two ways to give the names:

- We can use the standard R vector-definition function c( ) to define a character vector: Clusters = c("City", "postcode") Note the presence of quotation marks.
- We can use the special function clustgroup:

Clusters = clustgroup(City, postcode) Note the absence of quotation marks.

Either of these can be used, at the convenience of the user. Cluster variables are specified in order of level, i.e. the specification abovc says that City defines the level-1 clusters, and postcode the level-2 subclusters. For one level, only one cluster variable is needed, and clustgroup ( ) or c( ) can be omitted, as in the clusters = Litter. Each unique value of the cluster variable corresponds to one cluster. The cluster variable can be numeric (usually integer), or a factor, such as city names. There is no need for integer values to be consecutive: any set of numbers will do, such as U. S. postal codes ("zipcodes"), which are 5-digit numbers.

Type: Type of covariance model. Choices are listed above in the function definition (default first). FixRhoValueLev1: Numeric value to fix level 1 "rho" parameter in distance decay models; usually 0 if used.

FixRhoValueLev2: Numeric value to fix level 2 "rho" parameter; usually 0 if used .

RhoUpBndLev1, RhoUpBndLev2: Optional override for the program's calculation of the maximum value of the rho-parameter value in distance-decay models. Will rarely be used.

DistMatLev1: Data frame name, or filename and path, if needed, of level 1 distance matrix, in distance decay models.

DistMatLev2: Data frame name, or filename and path, if needed, of level 2 distance matrix.

DistMatTypeLev1, DistMatTypeLev2: At present, NUMERIC is the only type supported, so these arguments should be omitted.

DistScaleFactorLev1: Numeric scale factor for level 1 distance matrix (e.g. to convert distance units). Default 1.

DistScaleFactorLev2: Numeric scale factor for level 2 distance matrix. Default 1.

ClustWeightVecLev1: Level 1 weight vector: data frame or file name and path

ClustWeightVecLev2: Level 2 weight vector: data frame or file name and path.

SubclustersSimilar: In 2-level or multi-level model, should clusters all have the same subcluster parameters? Values: one of TRUE, FALSE. If FALSE, then each cluster has a cluster-specific set of values for the dispersion parameters of its subclusters. If TRUE, there is one set of subcluster dispersion parameters for all clusters.

Value returned: An object of class "RE f f Prop s", which can be specified to the CoxPoiss function.

clustgroup

Groups two or more cluster-variable names together for multi-level random effects models: clusters = clustgroup (x, y) is equivalent to using the standard R function clusters = c("x", "y"), and either can be used in defining the cluster hierarchy for the randEf fects function's c lu s te r s argument.

survProps

Groups the essential response information for a survival model: end time variable, event indicator, and some optional items.

survProps <- function(starttime = 0, endtime, timeorg = 0, event, EventIsZero = FALSE)

Arguments:

starttime: In a survival problem, each input record has a time interval associated with it. This argument is the name of the variable that gives the starting time of the interval. If not specified, then all intervals are considered to start at time 0, i.e. the intervals represent durations rather than calendar times.

endtime: Name of variable that gives the end time of the interval; required argument.

timeorg: Name of a variable that gives a time origin: will be subtracted from start and end times. Main use is to put repeated time intervals for the same subject on the same duration interval ("Gap-time" formulation).

event: Name of variable indicating whether or not an event occurs at the end-time for the record; required argument.

EventIsZero: By default, a non-zero value for the event-variable indicates an event, and a zero value indicates no event (i.e. censoring). Setting EventIsZero=TRUE reverses this convention, and makes a zero value indicate an event, and a non-zero value censoring.

Value returned: An object of class "SurvProps", which can be specified as the "response" part of a survival model formula.

describe.primary.data

If the data is to be read from a file, this gives characteristics of the primary data set. Unnecessary if the primary data is in a data frame. See the section "Large Data Files" for more information and some examples.

describe.primary.data <- function(dataSource, varnames = NULL, vartypes = NULL)

Arguments:

dataSource: data frame or filename or connection that contains the data.

varnames: A character vector of names of the variables in the data source.

vartypes: An R list of variable types, which are either "integer", "real", or factor descriptors as produced by the function "describe . factor". A synonym for "real" is "double".

As an alternative to using the arguments varnames and vartypes, the first two lines of the data file can contain names and types: but then the types must be numeric codes, 1 for "integer", and 2 for "real". Factors are not allowed when using this option, and all variables in the file must be numeric

Value returned: An object of class "DataSource", which can be specified as the "primary" argument in the functions CoxPoiss.

describe.secondary.data

A survival model can have a "secondary data set", described in the section on "Time-Dependent Covariates and Secondary Data Tables", and this function defines the data set for the program. Unnecessary if the secondary data set is in a data frame.

describe.secondary.data <- function(dataSource, varnames = NULL, vartypes = NULL, keyvarname, starttime = 0, timeorg = 0, endtime)

Arguments:

dataSource, varnames, vartypes: same as the corresponding arguments of the function describe.secondary.data.

keyvarname: name of the "key" variable, which has the same name in the primary and secondary data sets, and which serves to link records between the two.

starttime, timeorg, endtime: names of variables defining the time interval over which the secondary-table covariates are defined, for each record.

Value returned: An object of class c("DataSource" , "SecTabDescr"), which can be specified in the "secondary" argument of the function CoxPoiss.

describe.factor

If any covariates or other variables are "factors" in the R terminology, and if the primary data is in a file, then the factor's properties must be defined for the program. Unnecessary if the primary data is in a data frame. See the section "Large Data Files" for more information and examples.

describe.factor <- function(levelvals = NULL, levelnames = NULL, isordered = FALSE)

Arguments:

Levelvals: character or numeric vector. All the values that the factor variable has, as they appear in the data file.

Levelnames: character or numeric vector, corresponding to the Levelvals vector. The recoded or renamed values as they are to appear in the program output. Default is no renaming. Isordered: not used at present.

Value returned: Object of class "factor.descriptor", that can be included in a vartypes list.

RobustVariance

Specifies to the program that the standard errors of regression coefficients are to be computed using a "robust" variance estimator, and gives some grouping information.

RobustVariance <- function(UseRobust = FALSE, ShowRobust = FALSE, groupVarname = NULL, omitValue = NULL)

Arguments:

UseRobust: use robust variance to compute t-statistic and confidence intervals

ShowRobust: include robust variance in output listings

groupVarname: variable for collapsing to individuals (usually subject-ID) MUST BE INTEGER OR FACTOR. The grouping variable should define the groups within which observations are considered correlated, and between which they are independent.

omitValue: One value of the grouping variable (integer or factor): any record for which the grouping variable is equal to omitValue, will be ignored in the robust variance calculation. The argument omitValue is ignored if no grouping variable, or if the values of the grouping variable don't include the value omitValue.

Value returned: Object of class "RobustVariancePars", to be used in the "robustVarSpec" argument of the functions CoxPoiss.

CoxPoiss

This is the function that does the actual estimation of a survival model.

CoxPoiss <- function(model, primary, RunName = "CoxPoissRun", secondary = NULL, RandomEffects = NULL, strata = NULL, logFile = NULL, outheading = NULL, ByVariableSpec = NULL, subsetVariableSpec = NULL, robustVarSpec = NULL, covMatFile = NULL, RECovMatFile = NULL, maxiterations = 50, tolerance = 1.0e-6, SaveName = NULL)

Arguments:

model: model formula (required): the "response" part must include a "survProps" function

primary: "DataSource" object (required), as produced by the function describe.primary.data

RunName: Any string with no embedded blanks: appears in output with an appended code.

Secondary: object of class c("DataSource" , "SecTabDescr"), as produced by the function describe.secondary.data

RandomEffects: "REffProps" object, as produced by the function randEffects.

Strata: name of a variable in the data; each value of the variable defines a stratum.

LogFile: A string containing a path and filename, the name of the log file to be written. If omitted, no log file is written.

Outheading: Any string: will appear as the title of the output.

ByVariableSpec: Not used at present.

SubsetVariableSpec: Name of numeric variable in data: the data records with non-0 values of this variable will be selected for the analysis. Records with 0 will be omitted.

RobustVarSpec: "RobustVariancePars" object; as produced by the function RobustVariance

CovMatFile: string with filename and path: the covariance matrix of the estimated coefficients will be written to the file. However, the covariance matrix is also included in the results object produced by the CoxPoi s s function, and is probably more easily accessed in this way.

RECovMatFile: string with filename and path: the random effects covariance matrix at the finest level. Same remarks apply as for CovMatFile.

Maxiterations: positive integer.

Tolerance: positive real number.

SaveName: Filename (and path, if needed): if supplied, the results object produced by the CoxPoi s s will be saved in a file, and can be restored in a later session by the function Re s toreCoxP.

Note: there are a few other arguments accepted by the CoxPoiss function, but users will rarely if ever have occasion to use them.

Value returned: Object of class "CoxPoiss", or a list of such objects. If random effects are specified, the list will include two objects: the results without and with random effects.

summary.CoxPoiss

Prints on the screen, and optionally writes to a file, a summary description of the results contained in an object of class “CoxPoiss”.

summary.<class> <- function(object, printrandeff = FALSE, file = NULL, conflevel = .95, digits = 6) where .< c la s s> is one of the class suffixes.

Arguments:

object: the name of an object of class “CoxPoiss”, as produced by the corresponding function.

printrandeff: If TRUE, random effects values will be listed (which can be bulky).

file: If a filename (and path, if necessary) is given, the output will be written to the file instead of sown on the screen. If the file does not already exist it will be created. If it does, the previous contents will be lost.

conflevel: the desired confidence level for confidence intervals.

digits: number of significant figures to be displayed.

RestoreCoxP

Brings back an object of type “CoxPoiss” previously saved to disk, for further processing.

RestoreCoxP <- function(saveFileName, keepFile=TRUE)

Arguments:

saveFileName: the name (and path, if necessary) of the file into which the object was saved.

keepFile: if FALSE, the file will be deleted after restoring.