TRIMR — TRIM in R

Patrick Bogaart, Mark van der Loo, Jeroen Pannekoek, Arco van Strien

September 11, 2016

Contents

1 Parameter estimation

This Section describes the core TRIM function, which estimates the TRIM parameters.

First intriduce some helper functions.

```
> set_trim_verbose <- function(verbose=FALSE){
>   stopifnot(isTRUE(verbose)|!isTRUE(verbose))
>   options(trim_verbose=verbose)
> }
> set_trim_verbose(TRUE)
```

Convenience function for console output during runs

```
> rprintf <- function(fmt,...) { if(getOption("trim_verbose")) cat(sprintf(fmt,...)) }
```

Similar, but for object/summary printing

```
> printf <- function(fmt,...) {cat(sprintf(fmt,...))}
```

Let's het started with the main workhorse function.

```
> trim_estimate <- function(count, time.id, site.id, covars=list(),
>             model=2, serialcor=FALSE, overdisp=FALSE,
>             changepoints=integer(0)) {
```

1.1 Preparation

Check the arguments. `count` should be a vector of numerics.

```
>   stopifnot(class(count) %in% c("integer","numeric"))
>   n = length(count)
```

`time.id` should be an ordered factor, or a vector of consecutive years or numbers Note the use of "any" because of multiple classes for ordered factors

```
>   stopifnot(any(class(time.id) %in% c("integer","numeric","factor")))
>   if (any(class(time.id) %in% c("integer","numeric"))) {
>     check = unique(diff(sort(unique(time.id))))
>     stopifnot(check==1 && length(check)==1)
>   }
>   stopifnot(length(time.id)==n)
```

Convert the time points to a factor

`site.id` should be a vector of numbers, strings or factors

```
>   stopifnot(class(site.id) %in% c("integer","character","factor"))
>   stopifnot(length(site.id)==n)
```

`covars` should be a list where each element (if any) is a vector

```
>   stopifnot(class(covars)=="list")
>   ncovar = length(covars)
>   use.covars <- ncovar>0
```

```
>   if (use.covars) {
>     for (i in 1:ncovar) stopifnot(class(covars[[i]]) %in% c("integer","numeric"))
```

Also, each covariate i should be a number (ID) ranging $1 \ldots nclass_i$

```
>     nclass <- numeric(ncovar)
>     for (i in 1:ncovar) {
>       cv <- covars[[i]]  # The vector of covariate class ID's
>       stopifnot(min(cv)==1)  # Assert lower end of range
>       nclass[i] = max(cv)  # Upper end of range
>       stopifnot(nclass[i]>1)  # Assert upper end
>       stopifnot(length(unique(cv))==nclass[i])  # Assert the range is contiguous
>     }
>   }
```

`model` should be in the range 1 to 3

```
>   stopifnot(model %in% 1:3)
>   if (model==1){
>     message("Alas, Model 1 is not implemented yet. Returning zippedidooda (NULL)")
>     return(NULL)
>   }
```

Convert time and site to factors, if they're not yet

```
>   if (any(class(time.id) %in% c("integer","numeric"))) time.id <- ordered(time.id)
>   ntime = length(levels(time.id))
```

```
>   if (class(site.id) %in% c("integer","numeric")) site.id <- factor(site.id)
>   nsite = length(levels(site.id))
```

Create observation matrix f. Convert the data from a vector representation to a matrix representation. It's OK to have missing site/time combinations; these will automatically translate to NA values.

```
>   f <- matrix(0, nsite, ntime)  # ??? check if we shoud not use NA instead of 0!!!
>   rows <- as.integer(site.id)  # 'site.id' is a factor, thus this results in 1...I.
>   cols <- as.integer(time.id)  # idem, 1...J.
>   idx <- (cols-1)*nsite+rows  # Create column-major linear index from row/column subscripts.
>   f[idx] <- count  # ... such that we can paste all data into the right positions
```

Create similar matrices for all covariates

```
>   if (use.covars) {
>     cvmat <- list()
>     for (i in 1:ncovar) {
>       cv = covars[[i]]
>       m <- matrix(NA, nsite, ntime)
>       m[idx] <- cv
>       cvmat[[i]] <- m
>     }
>   }
```

We often need some specific subset of the data, e.g. all observations for site 3. These are conveniently found by combining the following indices:

```
>   observed <- is.finite(f)  # Flags observed (TRUE) / missing (FALSE) data
>   site <- as.vector(row(f))  # Internal site identifiers are the row numbers of the original matrix.
>   time <- as.vector(col(f))  # Idem for time points.
>   nobs <- rowSums(observed)  # Number of actual observations per site
```

For model 2, we do not allow for changepoints < 1 or $\geq J$. At the same time, a changepoint 1 must be present

```
>   if (model==2) {
>     if (length(changepoints)==0) {
>       use.changepoints <- FALSE  # Pretend we're not using changepoints at all
```

```
>       changepoints <- 1L  # but internally use them nevertheless
>     } else {
>       use.changepoints <- TRUE
>       stopifnot(all(changepoints>=1L))
>       stopifnot(all(changepoints<ntime))
>       stopifnot(all(diff(changepoints)>0))
>       if (changepoints[1]!=1L) changepoints = c(1L, changepoints)
>     }
>   }
```

For model 3, changepoints are not allowed TODO: proper error msg "No Changepoints allowed with model 3"

```
>   if (model==3) stopifnot(length(changepoints)==0)
```

We make use of the generic model structure

$$\log \mu = A\alpha + B\beta$$

where design matrices A and B both have IJ rows. For efficiency reasons the model estimation algorithm works on a per-site basis. There is thus no need to store these full matrices. Instead, B is constructed as a smaller matrix that is valid for any site, and A is not used at all.

Create matrix B, which is model-dependent.

```
>   if (model==2) {
>     ncp  <-  length(changepoints)
>     J <- ntime
>     B <- matrix(0, J, ncp)
>     for (i in 1:ncp) {
>       cp1  <-  changepoints[i]
>       cp2  <-  ifelse(i<ncp, changepoints[i+1], J)
>       if (cp1>1) B[1:(cp1-1), i]  <-  0
>       B[cp1:cp2,i]  <-  0:(cp2-cp1)
>       if (cp2<J) B[(cp2+1):J,i]  <-  B[cp2,i]
>     }
>   } else if (model==3) {
```

Model 3 in it's canonical form uses a single time parameter γ per time step, so design matrix B is essentially a $J\times$J identity matrix. Note, hoewever, that by definition $\gamma_1 = 0$, so effectively there are $J-1$ γ-values to consider. As a consequence, the first column is deleted.

```
>     B <- diag(ntime)  # Construct J×J identity matrix
>     B <- B[ ,-1]  # Remove first column
>   }
```

optionally add covariates. Each covar class (except class 1) adds an extra copy of B, where rows are cleared if that site/time combi does not participate in

```
>   if (use.covars) {
>     cvmask <- list()
>     for (cv in 1:ncovar) {
>       cvmask[[cv]] = list()
>       for (cls in 2:nclass[cv]) {
>         cvmask[[cv]][[cls]] <- list()
>         for (i in 1:nsite) {
>           cvmask[[cv]][[cls]][[i]] <- which(cvmat[[cv]][i, ]!=cls)
>         }
>       }
>     }
```

The amount of extra parameter sets is the total amount of covarariate classes minus the number of covariates (because class 1 does not add extra params)

```
>     num.extra.beta.sets <- sum(nclass-1)
>   }
```

When we use covariates, B is site-specific. We thus define a function to make the proper B for each site i

```
>   B0 <- B  # The "standard" B
>   rm(B)
>   make.B <- function(i, debug=FALSE) {
>     if (debug) { printf("make.B(%i): B0:", i); str(B0) }
>     if (model==2 || model==3) {
>       if (use.covars) {
```

Model 2 with covariates. Add a copy of B for each covar class

```
>         Bfinal <- B0
>         for (cv in ncovar) {
>           if (debug) printf("adding covar %d\n", cv)
>           for (cls in 2:nclass[cv]) {
>             if (debug) printf("adding class %d\n", cls)
>             Btmp <- B0
>             mask <- cvmask[[cv]][[cls]][[i]]
>             if (length(mask)>0) Btmp[mask, ] = 0
>             Bfinal <- cbind(Bfinal, Btmp)
>           }
>         }
>       } else {
```

Model 2 without covariates. Just the normal B

```
>         Bfinal = B0
>       }
>     } else if (model==3) {
>       Bfinal = B0
>     }
>     if (debug) { printf("make.B(%i): Bfinal:", i); str(Bfinal)}
>     Bfinal
>   }
```

1.2 Setup parameters and state variables

Parameter α has a unique value for each site.

```
>   alpha <- matrix(0, nsite,1)  # Store as column vector
```

Parameter β is model dependent.

```
>   if (model==2) {
```

For model 2 we have one β per change points

```
>     nbeta <- length(changepoints)
>   } else if (model==3) {
```

For model 3, we have one β per time $j > 1$

```
>     nbeta = ntime-1
>   }
```

If we have covariates, β's are repeated for each covariate class > 1.

```
>   nbeta0 <- nbeta  # Number of 'baseline' (i.e., without covariates) β's.
>   if (use.covars) {
>     nbeta <- nbeta0 * (sum(nclass-1)+1)
>   }
```

All β_j are initialized at 0, to reflect no trend (model 2) or no time effects (model 3)

```
>   beta <- matrix(0, nbeta,1)   # Store as column vector
```

Variable μ holds the estimated counts.

```
>   mu <- matrix(0, nsite, ntime)
```

1.3 Model estimation

TRIM estimates the model parameters α and β in an iterative fashion, so separate functions are defined for the updates of these and other variables needed.

```
>   # 3 Site-parameters α
```

Update α_i using:

$$\alpha_i^t = \log z_i' f_i - \log z_i' \exp(B_i \beta^{t-1})$$

where vector z contains just ones if autocorrelation and overdispersion are ignored (i.e., Maximum Likelihood, ML), or weights, when these are taken into account (i.e., Generalized Estimating Equations, GEE). In this case,

$$z = \mu V^{-1}$$

with V a covariance matrix (see Section 1.3.2).

```
>   update_alpha <- function(method=c("ML","GEE")) {
>     for (i in 1:nsite) {
>       B = make.B(i)
>       f_i <- f[site==i & observed==TRUE]   # vector
>       B_i <- B[observed[site==i], , drop=FALSE]
>       if (method=="ML") {   # no covariance; V_i = diag(mu)
>         z_t <- matrix(1, 1, nobs[i])
>       } else if (method=="GEE") {   # Use covariance
>         mu_i = mu[site==i & observed==TRUE]
>         z_t <- mu_i %*% V_inv[[i]]   # define correlation weights
>       } else stop("Can't happen")
>       alpha[i] <<- log(z_t %*% f_i) - log(z_t %*% exp(B_i %*% beta))
>     }
>   }
```

1.3.1 Time parameters β

Estimates for parameters β are improved by computing a change in β and adding that to the previous values:

$$\beta^t = \beta^{t-1} - (i_b)^{-1} U_b^*$$

where i_b is a derivative matrix (see Section 1.3.5) and U_b^* is a Fisher Scoring matrix (see Section 1.3.5). Note that the 'improvement' as defined by (1.3.1) can actually results in a decrease in model fit. These cases are identified by measuring the model Likelihood Ratio (Eqn (22)). If this measure increases, then smaller adjustment steps are applied. This process is repeated until an actually improvement is found.

```
>   update_beta <- function(method=c("ML","GEE"))
>   {
>     update_U_i()   # update Score U_b and Fisher Information i_b
```

Compute the proposed change in β.

```
>     dbeta  <-  -solve(i_b) %*% U_b
```

This is the maximum update; if it results in an *increased* likelihood ratio, then we have to take smaller steps. First record the original state and likelihood.

```
>     beta0 <- beta
>     lik0  <- likelihood()
```

```
>     stepsize = 1.0
>     for (subiter in 1:7) {
>       beta <<- beta0 + stepsize*dbeta
>       update_mu(fill=FALSE)
>       update_alpha(method)
>       update_mu(fill=FALSE)
>       lik <- likelihood()
>       if (lik < lik0) break else stepsize <- stepsize / 2  # Stop or try again
>     }
>     subiter
>   }
```

1.3.2 Covariance and autocorrelation

Covariance matrix V_i is defined by

$$V_i = \sigma^2 \sqrt{\text{diag}(\mu)} R \sqrt{\text{diag}(\mu)} \tag{1}$$

where σ^2 is a dispersion parameter (Section 1.3.3) and R is an (auto)correlation matrix. Both of these two elements are optional. If the counts are perfectly Possion distributed, $\sigma^2 = 1$, and if autocorrelation is disabled (i.e. counts are independent), Eqn (1) reduces to

$$V_i = \sigma^2 \, \text{diag}(\mu) \tag{2}$$

```
>   V_inv  <- vector("list", nsite)  # Create storage space for V_i^{-1}.
>   Omega <- vector("list", nsite)
>   update_V <- function(method=c("ML","GEE")) {
>     for (i in 1:nsite) {
>       mu_i <- mu[site==i & observed]
>       f_i  <- f[site==i & observed]
>       d_mu_i <- diag(mu_i, length(mu_i))  # Length argument guarantees diag creation
>       if (method=="ML") {
>         V_i <- sig2 * d_mu_i
>       } else if (method=="GEE") {
>         idx <- which(observed[i, ])
>         R_i <- Rg[idx,idx]
>         V_i <- sig2 * sqrt(d_mu_i) %*% R_i %*% sqrt(d_mu_i)
>       } else stop("Can't happen")
>       V_inv[[i]] <<- solve(V_i) # Store $V^{-1}  # for later use
>       Omega[[i]] <<- d_mu_i %*% V_inv[[i]] %*% d_mu_i  # idem for Ω_i
>     }
>   }
```

The (optional) autocorrelation structure for any site i is stored in $n_i \times n_i$ matrix R_i. In case there are no missing values, $n_i = J$, and the 'full' or 'generic' autocorrelation matrix R is expressed as

$$R = \begin{pmatrix} 1 & \rho & \rho^2 & \cdots & \rho^{J-1} \\ \rho & 1 & \rho & \cdots & \rho^{J-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \rho^{J-1} & \rho^{J-2} & \rho^{J-3} & \cdots & 1 \end{pmatrix} \tag{3}$$

where ρ is the lag-1 autocorrelation.

```
>   Rg <- diag(1, ntime)  # default (no autocorrelation) value
>   update_R <- function() {
>     Rg <<- rho ^ abs(row(diag(ntime)) - col(diag(ntime)))
>   }
```

Lag-1 autocorrelation parameter ρ is estimated as

$$\hat{\rho} = \frac{1}{n_{i,j,j+1}\hat{\sigma}^2} \left(\sum_{i}^{I} \sum_{j}^{J-1} r_{i,j} r_{i,j+1}) \right) \tag{4}$$

where the summation is over observed pairs i, j–$i, j{+}1$, and $n_{i,j,j+1}$ is the number of pairs involved. Again, both ρ and R are computes ρ in a stepwise per-site fashion. Also, site-specific autocorrelation matrices R_i are formed by removing the rows and columns from R corresponding with missing observations.

```
>   rho <- 0.0  # default value (ML)
>   update_rho <- function() {
```

First estimate ρ

```
>     rho   <-  0.0
>     count <-  0
>     for (i in 1:nsite) {
>       for (j in 1:(ntime-1)) {
>         if (observed[i,j] && observed[i,j+1]) {  # short-circuit AND intended
>           rho <- rho + r[i,j] * r[i,j+1]
>           count <- count+1
>         }
>       }
>     }
>     rho <<- rho / (count * sig2)  # compute and store in outer environment
>   }
```

1.3.3 Overdispersion.

Dispersion parameter σ^2 is estimated as

$$\hat{\sigma}^2 = \frac{1}{n_f - n_\alpha - n_\beta} \sum_{i,j} r_{ij}^2 \tag{5}$$

where the n terms are the number of observations, α's and β's, respectively. Summation is over the observed i, j only. and r_{ij} are Pearson residuals (Section 1.3.4)

```
>   sig2 <- 1.0  # default value (Maximum Likelihood case)
>   update_sig2 <- function() {
>     df <- sum(nobs) - length(alpha) - length(beta)  # degrees of freedom
>     sig2 <<- sum(r^2, na.rm=TRUE) / df
>   }
```

1.3.4 Pearson residuals

Deviations between measured and estimated counts are quantified by the Pearson residuals r_{ij}, given by

$$r_{ij} = (f_{ij} - \mu_{ij})/\sqrt{\mu_{ij}} \tag{6}$$

```
>   r <- matrix(0, nsite, ntime)
>   update_r <- function() {
>     r[observed] <<- (f[observed]-mu[observed]) / sqrt(mu[observed])
>   }
```

1.3.5 Derivatives and GEE scores

Derivative matrix i_b is defined as

$$-i_b = \sum_i B_i' \left(\Omega_i - \frac{1}{d_i} \Omega_i z_i z_i' \Omega_i \right) B_i \tag{7}$$

where

$$\Omega_i = \mathrm{diag}(\mu_i) V_i^{-1} \mathrm{diag}(\mu_i) \tag{8}$$

with V_i the covariance matrix for site i, and

$$d_i = z_i' \Omega_i z_i \tag{9}$$

```
>   i_b <- 0
>   U_b <- 0
>   update_U_i <- function() {
>     i_b <<- 0  # Also store in outer environment for later retrieval
>     U_b <<- 0
>     for (i in 1:nsite) {
>       B = make.B(i)
>       mu_i <- mu[site==i & observed]
>       f_i  <- f[site==i & observed]
>       d_mu_i <- diag(mu_i, length(mu_i))  # Length argument guarantees diag creation
>       ones <- matrix(1, nobs[i], 1)
>       d_i <- as.numeric(t(ones) %*% Omega[[i]] %*% ones)  # Could use sum(Omega) as well...
>       B_i <- B[observed[site==i], ,drop=FALSE]
>       i_b <<- i_b - t(B_i) %*% (Omega[[i]] - (Omega[[i]] %*% ones %*% t(ones) %*% Omega[[i]]) / d_
>       U_b <<- U_b + t(B_i) %*% d_mu_i %*% V_inv[[i]] %*% (f_i - mu_i)
>     }
>   }
```

1.3.6 Count estimates

Let's not forget to provide a function to update the modelled counts μ_{ij}:

$$\mu^t = \exp(A\alpha^t + B\beta^{t-1} - \log w)$$

where it is noted that we do not use matrix A. Instead, the site-specific parameters α_i are used directly:

$$\mu_i^t = \exp(\alpha_i^t + B\beta^{t-1} - \log w)$$

```
>   update_mu <- function(fill) {
>     for (i in 1:nsite) {
>       B = make.B(i)
>       mu[i, ] <<- exp(alpha[i] + B %*% beta)
>     }
```

clear estimates for non-observed cases, if required.

```
>     if (!fill) mu[!observed] <<- 0.0
>   }
```

1.3.7 Likelihood

```
>   likelihood <- function() {
>     lik <- 2*sum(f*log(f/mu), na.rm=TRUE)
>     lik
>   }
```

1.3.8 Convergence

The parameter estimation algorithm iterated until convergence is reached. 'convergence' here is defined in a multivariate way: we demand convergence in model paramaters α and β, model estimates μ and likelihood measure L.

```
>   new_par <- new_cnt <- new_lik <- NULL
>   old_par <- old_cnt <- old_lik <- NULL
>   check_convergence <- function(iter, crit=1e-5) {
```

Collect new data for convergence test (Store in outer environment to make them persistent)

```
>     new_par <<- c(as.vector(alpha), as.vector(beta))
>     new_cnt <<- as.vector(mu)
>     new_lik <<- likelihood()

>     if (iter>1) {
>       max_par_change <- max(abs(new_par - old_par))
>       max_cnt_change <- max(abs(new_cnt - old_cnt))
>       max_lik_change <- max(abs(new_lik - old_lik))
>       conv_par <- max_par_change < crit
>       conv_cnt <- max_cnt_change < crit
>       conv_lik <- max_lik_change < crit
>       convergence <- conv_par && conv_cnt && conv_lik
>       rprintf(" Max change: %10e %10e %10e ", max_par_change, max_cnt_change, max_lik_change)
>     } else {
>       convergence = FALSE
>     }
```

Today's new stats are tomorrow's old stats

```
>     old_par <<- new_par
>     old_cnt <<- new_cnt
>     old_lik <<- new_lik

>     convergence
>   }
```

1.3.9 Main estimation procedure

Now we have all the building blocks ready to start the iteration procedure. We start 'smooth', with a couple of Maximum Likelihood iterations (i.e., not considering $\sigma^2 \neq 1$ or $\rho > 0$), after which we move to on GEE iterations if requested.

```
>   method     <- "ML"  # start with Maximum Likelihood
>   final_method <- ifelse(serialcor || overdisp, "GEE", "ML")  # optionally move on to GEE

>   max_iter  <- 100  # Maximum number of iterations allowed
>   conv_crit <- 1e-7
>   for (iter in 1:max_iter) {
>     rprintf("Iteration %d (%s)", iter, method)

>     update_alpha(method)
>     update_mu(fill=FALSE)
>     if (method=="GEE") {
>       update_r()
>       if (overdisp)  update_sig2()
>       if (serialcor) update_rho()
>       update_R()
>     }
>     update_V(method)
>     subiters <- update_beta(method)
>     rprintf(", %d subiters", subiters)
>     rprintf(", lik=%.3f", likelihood())
>     if (overdisp)  rprintf(", sig^2=%.5f", sig2)
>     if (serialcor) rprintf(", rho=%.5f;", rho)

>     convergence <- check_convergence(iter)
```

```
>     if (convergence && method==final_method) {
>       rprintf("\nConvergence reached\n")
>       break
>     } else if (convergence) {
>       rprintf("\nChanging ML --> GEE\n")
>       method = "GEE"
>     } else {
>       rprintf("\n")
>     }
>   }
```

If we reach the preset maximum number of iterations, we clearly have not reached convergence.

```
>   if (iter==max_iter) stop("No convergence reached.")
```

Run the final model

```
>   update_mu(fill=TRUE)
```

Covariance matrix

```
>   var_beta <- -solve(i_b)
```

1.4 Imputation

The imputation process itself is trivial: just replace all missing observations $f_{i,j}$ by the model-based estimates $\mu_{i,j}$.

```
>   imputed <- ifelse(observed, f, mu)
```

1.5 Output and postprocessing

Measured, modelled and imputed count data are stored in a TRIM output object, together with parameter values and other usefull information.

```
>   z <- list(title=title, f=f, nsite=nsite, ntime=ntime, nbeta0=nbeta0,
>             covars=covars, ncovar=ncovar,
>             model=model, changepoints=changepoints,
>             mu=mu, imputed=imputed, alpha=alpha, beta=beta, var_beta=var_beta)
>   if (use.covars) {
>     z$ncovar <- ncovar  # todo: eliminate?
>     z$nclass <- nclass
>   }
>   class(z) <- "trim"
```

Several kinds of statistics can now be computed, and added to this output object.

1.5.1 Overdispersion and Autocorrelation

```
>   z$sig2 <- ifelse(overdisp, sig2, NA)
>   z$rho  <- ifelse(serialcor, rho,  NA)
```

1.5.2 Coefficients and uncertainty

```
>   if (model==2) {
>     se_beta  <- sqrt(diag(var_beta))
>     ncp = length(changepoints)
>     coefs = data.frame(
>       from   = changepoints,
```

```
>       upto   = if (ncp==1) ntime else c(changepoints[2:ncp], ntime),
>       add    = beta,
>       se_add = se_beta,
>       mul    = exp(beta),
>       se_mul = exp(beta) * se_beta
>     )
```

```
>     if (use.covars) {
```

Add some prefix columns with covariate and factor ID Note that we have to specify all covariate levels here, to prevent them from being to converted to NA later

```
>       prefix <- data.frame(covar = factor("baseline",levels=c("baseline", names(covars))),
>                            cat   = 0)
>       coefs <- cbind(prefix, coefs)
>       idx = 1:nbeta0
>       for (i in 1:ncovar) {
>         for (j in 2:nclass[i]) {
>           idx <- idx + nbeta0
>           coefs$covar[idx]  <- names(covars)[i]
>           coefs$cat[idx]    <- j
>         }
>       }
>     }
```

```
>     z$coefficients <- coefs
>   }  # if model==2
```

```
>   if (model==3) {
```

Model coefficients are output in two types; as additive parameters:

$$\log \mu_{ij} = \alpha_i + \gamma_j$$

and as multiplicative parameters:

$$\mu_{ij} = a_i g_j$$

where $a_i = e^{\alpha_i}$ and $g_j = e^{\gamma_j}$.

For the first time point, $\gamma_1 = 0$ by definition. So we have to add values of 0 for the baseline case and each covariate category > 1, if any.

```
>     #gamma <- matrix(beta, nrow=nbeta0)  # Each covariate category in a column
>     #gamma <- rbind(0, gamma)  # Add row of 0's for first time point
>     #gamma <- matrix(gamma, ncol=1)  # Cast back into a column vector
>     gamma <- beta
>     g     <- exp(gamma)
```

Parameter uncertainty is expressed as standard errors. For the additive parameters γ, the variance is estimated as

$$\mathrm{var}(\gamma) = (-i_b)^{-1}$$

```
>     var_gamma <-  -solve(i_b)
```

Finally, we compute the standard error as $\mathrm{S.E.}(\gamma) = \sqrt{\mathrm{diag}(\mathrm{var}(\gamma))}$

```
>     se_gamma  <-  sqrt(diag(var_gamma))
```

The standard error of the multiplicative parameters g_j is opproximated by using the delta method, which is based on a Taylor expansion:

$$\mathrm{var}(f(\theta)) = (f'(\theta))^2 \,\mathrm{var}(\theta) \tag{10}$$

which for $f(\theta) = e^\theta$ translates to

$$\mathrm{var}(g) = \mathrm{var}(e^\gamma) = e^{2\gamma}\,\mathrm{var}(\gamma)$$

leading to

$$\mathrm{S.E.}(g) = e^\gamma\,\mathrm{S.E.}(\gamma) = g\,\mathrm{S.E.}(\gamma)$$

```
>     se_g <- g * se_gamma
```

Baseline coefficients. Note that, because $\gamma_1 \equiv 0$, it was not estimated, and as a results $j = 1$ was not incuded in i_b, nor in var($gamma$) as computed above. We correct this by adding the 'missing' 0 (or 1 for multiplicative parameters) during output

```
>     idx = 1:nbeta0
>     coefs <- data.frame(
>       time   = 1:ntime,
>       add    = c(0, gamma[idx]),
>       se_add = c(0, se_gamma[idx]),
>       mul    = c(1, g[idx]),
>       se_mul = c(0, g[idx] * se_gamma[idx])
>     )
```

Covariate categories (> 1)

```
>     if (use.covars) {
>       prefix = data.frame(covar="baseline", cat=0)
>       coefs <- cbind(prefix, coefs)
>       for (i in 1:ncovar) {
>         for (j in 2:nclass[i]) {
>           idx <- idx + nbeta0
>           df <- data.frame(
>             covar  = names(covars)[i],
>             cat    = j,
>             time   = 1:ntime,
>             add    = c(0, gamma[idx]),
>             se_add = c(0, se_gamma[idx]),
>             mul    = c(1, g[idx]),
>             se_mul = c(0, g[idx] * se_gamma[idx])
>           )
>           coefs <- rbind(coefs, df)
>         }
>       }
>     }
>     z$coefficients <- coefs
>   }  # if model==3
```

1.5.3 Time totals

Recompute Score matrix i_b with final μ's

```
>   ib <- 0
>   for (i in 1:nsite) {
>     B <- make.B(i)
>     mu_i <- mu[site==i & observed]
>     n_i <- length(mu_i)
>     d_mu_i <- diag(mu_i, n_i)  # Length argument guarantees diag creation
>     OM <- Omega[[i]]
>     d_i <- sum(OM)  # equivalent with z' Omega z, as in the TRIM manual
>     B_i <- B[observed[site==i], ,drop=FALSE]
>     om <- colSums(OM)
>     OMzzOM <- om %*% t(om)  # equivalent with OM z z' OM, as in the TRIM manual
>     term <- t(B_i) %*% (OM - (OMzzOM) / d_i) %*% B_i
>     ib <- ib - term
>   }
```

Matrices E and F take missings into account

```
>   E <- -ib
>   nbeta <- length(beta)
>   F <- matrix(0, nsite, nbeta)
>   d <- numeric(nsite)
>   for (i in 1:nsite) {
>     B <- make.B(i)
>     d[i] <- sum(Omega[[i]])
>     w_i <- colSums(Omega[[i]])
>     B_i <- B[observed[site==i], ,drop=FALSE]
>     F_i <- (t(w_i) %*% B_i) / d[i]
>     F[i, ] <- F_i
>   }
```

Matrices G and H are for all mu's

```
>   GddG <- matrix(0, ntime,ntime)
>   for (i in 1:nsite) {
>     for (j in 1:ntime) for (k in 1:ntime) {
>       GddG[j,k] <- GddG[j,k] + mu[i,j]*mu[i,k]/d[i]
>     }
>   }
>   GF <- matrix(0, ntime, nbeta)
>   for (i in 1:nsite) {
>     for (j in 1:ntime) for (k in 1:nbeta)  {
>       GF[j,k] <- GF[j,k] + mu[i,j] * F[i,k]
>     }
>   }
>   H <- matrix(0, ntime, nbeta)
>   for (i in 1:nsite) {
>     B <- make.B(i)
>     for (k in 1:nbeta) for (j in 1:ntime) {
>       H[j,k]  <- H[j,k] + B[j,k] * mu[i,j]
>     }
>   }
>   GFminH <- GF - H
```

All building blocks are ready. Use them to compute the variance

```
>   var_tt_mod <- GddG + GFminH %*% solve(E) %*% t(GFminH)
```

To compute the variance of the time totals of the imputed data, we first substract the contribution due tp te observations, as computed by above scheme, and recplace it by the contribution due to the observations, as resulting from the covariance matrix.

```
>   muo = mu   # 'observed' μ's
>   muo[!observed] = 0 #   # erase estimated μ's
>   GddG <- matrix(0, ntime,ntime)
>   for (i in 1:nsite) if (nobs[i]>0) {
>     for (j in 1:ntime) for (k in 1:ntime) {
>       GddG[j,k] <- GddG[j,k] + muo[i,j]*muo[i,k]/d[i]
>     }
>   }
>   GF <- matrix(0, ntime, nbeta)
>   for (i in 1:nsite) if (nobs[i]>0) {
>     for (j in 1:ntime) for (k in 1:nbeta)  {
>       GF[j,k] <- GF[j,k] + muo[i,j] * F[i,k]
>     }
>   }
```

```
>   H <- matrix(0, ntime, nbeta)
>   for (i in 1:nsite) if (nobs[i]>0) {
>     B <- make.B(i)
>     for (k in 1:nbeta) for (j in 1:ntime) {
>       H[j,k]  <- H[j,k] + B[j,k] * muo[i,j]
>     }
>   }

>   GFminH <- GF - H

>   var_tt_obs_old <- GddG + GFminH %*% solve(E) %*% t(GFminH)
```

Now compute the variance due to observations

```
>   var_tt_obs_new = matrix(0, ntime, ntime)
>   for (i in 1:nsite) {
>     if (serialcor) {
>       srdu = sqrt(diag(muo[i, ]))
>       V = sig2 * srdu %*% Rg %*% srdu
>     } else {
>       V = sig2 * diag(muo[i, ])
>     }
>     var_tt_obs_new = var_tt_obs_new + V
>   }
```

Combine

```
>   var_tt_imp = var_tt_mod - var_tt_obs_old + var_tt_obs_new
```

Time totals of the model, and it's standard error

```
>   tt_mod    <- colSums(mu)
>   se_tt_mod <- round(sqrt(diag(var_tt_mod)))

>   tt_imp     <- colSums(imputed)
>   se_tt_imp <- round(sqrt(diag(var_tt_imp)))
```

Store in TRIM output

```
>   z$tt_mod <- tt_mod
>   z$tt_imp <- tt_imp
>   z$var_tt_mod <- var_tt_mod
>   z$var_tt_imp <- var_tt_imp

>   z$time.totals <- data.frame(
>     time    = 1:ntime,
>     model   = round(tt_mod),
>     se_mod  = se_tt_mod,
>     imputed = round(tt_imp),
>     se_imp  = se_tt_imp
>   )
```

1.5.4 Reparameterisation of Model 3

Here we consider the reparameterization of the time-effects model in terms of a model with a linear trend and deviations from this linear trend for each time point. The time-effects model is given by

$$\log \mu_{ij} = \alpha_i + \gamma_j, \tag{11}$$

with γ_j the effect for time j on the log-expected counts and $\gamma_1 = 0$. This reparameterization can be expressed as

$$\log \mu_{ij} = \alpha_i^* + \beta^* d_j + \gamma_j^*, \tag{12}$$

with $d_j = j - \bar{j}$ and $\bar{j}$ the mean of the integers j representing the time points. The parameter α_i^* is the intercept and the parameter β^* is the slope of the least squares regression line through the J log-expected

time counts in site i and γ_j^* can be seen as the residuals of this linear fit. From regression theory we have that the 'residuals"' γ_j^* sum to zero and are orthogonal to the explanatory variable, i.e.

$$\sum_j \gamma_j^* = 0 \quad \text{and} \quad \sum_j d_j \gamma_j^* = 0. \tag{13}$$

Using these constraints we obtain the equations:

$$\log \mu_{ij} = \alpha_i^* + \beta^* d_j + \gamma_j^* = \alpha_i + \gamma_j \tag{14}$$

$$\sum_j \log \mu_{ij} = J\alpha_j^* = J\alpha_i + \sum_j \gamma_j \tag{15}$$

$$\sum_j d_j \log \mu_{ij} = \beta^* \sum_j d_j^2 = \sum_j d_j \gamma_j, \tag{16}$$

where (14) is the re-parameterization equation itself and (15) and (16) are obtained by using the constraints (13)

From (15) we have that $\alpha_i^* = \alpha_i + \frac{1}{J}\sum_j \gamma_j$. Now, by using the equations (14) thru (16) and defining $D = \sum_j d_j^2$, we can express the parameters β^* and γ^* as functions of the parameters γ as follows:

$$\beta^* = \frac{1}{D}\sum_j d_j \gamma_j, \tag{17}$$

$$\begin{aligned}
\gamma_j^* &= \alpha_i + \gamma_j - \alpha_i^* - \beta^* d_j \quad \text{(using (5))} \\
&= \alpha_i - \left(\alpha_i + \frac{1}{J}\sum_j \gamma_j\right) + \gamma_j - d_j \frac{1}{D}\sum_j d_j \gamma_j \\
&= \gamma_j - \frac{1}{J}\sum_j \gamma_j - d_j \frac{1}{D}\sum_j d_j \gamma_j.
\end{aligned} \tag{18}$$

Since β^* and γ_j^* are linear functions of the parameters γ_j they can be expressed in matrix notation by

$$\begin{pmatrix} \beta^* \\ \boldsymbol{\gamma}^* \end{pmatrix} = \mathbf{T}\boldsymbol{\gamma}, \tag{19}$$

with $\boldsymbol{\gamma}^* = (\gamma_1^*, \ldots, \gamma_J^*)^T$, $\boldsymbol{\gamma} = (\gamma_1, \ldots, \gamma_J)^T$ and $\mathbf{T}$ the $(J+1) \times J$ transformation matrix that transforms $\boldsymbol{\gamma}$ to $\left(\beta^*, (\boldsymbol{\gamma}^*)^T\right)^T$. From (17) and (18) it follows that the elements of $\mathbf{T}$ are given by:

$$\begin{aligned}
\mathbf{T}_{(1,j)} &= \frac{d_j}{D} && (i = 1, j = 1, \ldots, J) \\
\mathbf{T}_{(i,j)} &= 1 - \frac{1}{J} - \frac{1}{D} d_{i-1} d_j && (i = 2, \ldots, J+1, j = 1, \ldots, J, i-1 = j) \\
\mathbf{T}_{(i,j)} &= -\frac{1}{J} - \frac{1}{D} d_{i-1} d_j && (i = 2, \ldots, J+1, j = 1, \ldots, J, i-1 \neq j)
\end{aligned}$$

```
>   if (model==3 && !use.covars) {
>     TT <- matrix(0, ntime+1, ntime)
>     J <- ntime
>     j <- 1:J; d <- j - mean(j)   # i.e, d_j = j - (1/J) sum_j j
>     D <- sum(d^2)   # i.e., D = sum_j d_j^2
>     TT[1, ] <- d / D
>     for (i in 2:(J+1)) for (j in 1:J) {
>       if (i-1 == j) {
>         TT[i,j] <- 1 - (1/J) - d[i-1]*d[j]/D
>       } else {
>         TT[i,j] <-   - (1/J) - d[i-1]*d[j]/D
>       }
>     }
```

```
>     gstar <- TT %*% c(0, gamma)  # Add the implicit γ1 = 0
>     bstar <- gstar[1]
>     gstar <- gstar[2:(J+1)]
```

The covariance matrix of the transformed parameter vector can now be obtained from the covariance matrix $\mathbf{T}\boldsymbol{\gamma}$ of $\boldsymbol{\gamma}$ as

$$V\begin{pmatrix}\beta^* \\ \boldsymbol{\gamma}^*\end{pmatrix} = \mathbf{T}V(\boldsymbol{\gamma})\mathbf{T}^T \tag{20}$$

```
>     var_gstar <- TT %*% rbind(0,cbind(0,var_gamma)) %*% t(TT)  # Again, γ1 = 0
>     se_bstar  <- sqrt(diag(var_gstar))[1]
>     se_gstar  <- sqrt(diag(var_gstar))[2:(ntime+1)]

>     z$gstar <- gstar
>     z$var_gstar <- var_gstar

>     z$linear.trend <- data.frame(
>       Additive       = bstar,
>       std.err        = se_bstar,
>       Multiplicative = exp(bstar),
>       std.err.       = exp(bstar) * se_bstar,
>       row.names      = "Slope",
>       check.names    = FALSE)
```

Deviations from the linear trend

```
>     z$deviations <- data.frame(
>       Time       = 1:ntime,
>       Additive   = gstar,
>       std.err.   = se_gstar,
>       Multiplicative = exp(gstar),
>       std.err.   = exp(gstar) * se_gstar,
>       check.names = FALSE
>     )

>   }
```

1.6 Return results

The TRIM result is returned to the user...

```
>   printf("(Exiting workhorse function)\n")
>   z
> }
```

...which ends the main TRIM function.

2 Goodness of fit

The goodness-of-fit of the model is assessed using three statistics: Chi-squared, Likelihood Ratio and Aikaike Information Content.

2.1 Computation

Here we define 'gof' as a S3 generic function

```
> gof <- function(x) UseMethod("gof")
```

Here is a simple wrapper function for TRIM output lists.

```
> gof.trim <- function(x) {
>   printf("gof.trim() called\n")
>   stopifnot(class(x)=="trim")
>   gof.numeric(x$f, x$mu, x$alpha, x$beta)
> }
```

Here is the workhorse function

```
> gof.numeric <- function(f, mu, alpha, beta) {
>   printf("gof.numeric() called\n")
>   observed <- is.finite(f)
```

The χ^2 (Chi-square) statistic is given by

$$\chi^2 = \sum_{ij} \frac{f_{i,j} - \mu_{i,j}}{\mu_{i,j}} \tag{21}$$

where the summation is over the observed i, j's only. Significance is assessed by comparing against a χ^2 distribution with *df* degrees of freedom, equal to the number of observations minus the total number of parameters involved, i.e. $df = n_f - n_\alpha - n_\beta$.

```
>   chi2 <- sum((f-mu)^2/mu, na.rm=TRUE)
>   df   <- sum(observed) - length(alpha) - length(beta)
>   p    <-  1 - pchisq(chi2, df=df)
>   chi2 <- list(chi2=chi2, df=df, p=p)  # store in a list
```

Similarly, the *Likelihood ratio* (LR) is computed as

$$\mathrm{LR} = 2 \sum_{ij} f_{ij} \log \frac{f_{i,j}}{\mu_{i,j}} \tag{22}$$

and again compared against a χ^2 distribution.

```
>   LR <- 2 * sum(f * log(f / mu), na.rm=TRUE)
>   df <- sum(observed) - length(alpha) - length(beta)
>   p  <- 1 - pchisq(LR, df=df)
>   LR <- list(LR=LR, df=df, p=p)
```

The Akaike Information Content (AIC) is related to the LR as:

```
>   AIC <- LR$LR - 2*LR$df
```

Output all goodness-of-fit measures in a single list

```
>   structure(list(chi2=chi2, LR=LR, AIC=AIC), class="trim.gof")
> }
```

2.2 Printing

A simple printing function is provided that mimics TRIM for Windows output.

```
> print.trim.gof <- function(x) {
>   stopifnot(class(x)=="trim.gof")
```

print welcome message

```
>   cat(sprintf("Goodness of fit\n"))
```

print χ^2 results

```
>   with(x$chi2,
>        printf("%24s = %.2f, df=%d, p=%.4f\n", "Chi-square", chi2, df, p))
```

idem, Likelihood ratio

```
>   with(x$LR,
>        printf("%24s = %.2f, df=%d, p=%.4f\n", "Likelihood Ratio", LR, df, p))
```

idem, Akaike Information Content

```
>   with(x,
>        printf("%24s = %.2f\n", "AIC (up to a constant)", AIC))
> }
```

3 TRIM postprocessing functions

3.1 Data summary

3.1.1 extract

```
> summary.TRIMdata <- function(x)
> {
>   stopifnot(class(x)=="TRIMdata")
```

Collect covariate data

```
>   covar_cols <- ifelse(x$weight, 5,4) : ncol(x$df)
>   covars <- data.frame(col = covar_cols,
>                        name = names(x$df)[covar_cols],
>                        levels = sapply(x$df[covar_cols], nlevels)
>   )
```

Create summary output

```
>   out <- list(ncols=ncol(x$df), file=x$file, nsite=x$nsite, ntime=x$ntime,
>               missing=x$missing, covars=covars,
>               nzero=x$nzero, npos=x$npos, nobs=x$nobs, nmis=x$nmis,
>               ncount=x$ncount, totcount=x$totcount)
>   class(out) <- "summary.TRIMdata"
>   out
> }
```

```
> dominant_sites <- function(x, threshold=10) {
>   stopifnot(class(x)=="TRIMdata")
```

Compute site totals

```
>   ST <- ddply(x$df, .(site), summarize, total=sum(count, na.rm=TRUE))
>   ST$percent <- 100 * ST$total / x$totcount
```

Dominate sites: more than 10>   DOM <- subset(ST, percent>threshold)

Create summary output

```
>   out <- list(sites=DOM, threshold=threshold)
>   class(out) <- "trim.dom"
>   out
> }
```

```
> average <- function(x)
> {
>   stopifnot(class(x)=="TRIMdata")
```

Number of observations and mean count for each time point can be computed directly using the plyr tools.

```
>   out <- ddply(x$df, .(time), summarise,
>                observations=sum(is.finite(count)),
>                average=mean(count, na.rm=TRUE))
```

```
>   out$index <- out$average / out$average[1]
>   out
> }
```

3.1.2 print

```
> print.summary.TRIMdata <- function(x)
> {
>   stopifnot(class(x)=="summary.TRIMdata")
>   printf("\nThe following %d variables have been read from file: %s\n", x$ncols, x$file)
>   printf("1. %-20s number of values: %d\n", "Site", x$nsite)
>   printf("2. %-20s number of values: %d\n", "Time", x$ntime)
>   printf("3. %-20s missing = %d\n", "Count", x$missing)
```

TODO: weight column

```
>   for (i in 1:nrow(x$covars)) {
>     printf("%d. %-20s number of values: %d\n", x$covars$col[i], x$covars$name[i], x$covars$levels[
>   }
>   printf("\n")
>   printf("Number of observed zero counts      %8d\n", x$nzero)
>   printf("Number of observed positive counts  %8d\n", x$npos)
>   printf("Total number of observed counts     %8d\n", x$nobs)
>   printf("Number of missing counts            %8d\n", x$nmis)
>   printf("Total number of counts              %8d\n", x$ncount)
>   printf("\n")
>   printf("Total count                         %8d\n", x$totcount)
> }
> print.trim.dom <- function(dom) {
>   stopifnot(class(dom)=="trim.dom")
>   printf("\nSites containing more than %d%% of the total count:\n", dom$threshold)
>   print(dom$sites, row.names=FALSE)
> }
```

3.2 Summary

```
> summary.trim <- function(x) {
>   stopifnot(class(x)=="trim")
>   if (is.finite(x$sig2) || is.finite(x$rho)) {
>     out = list(est.method="Generalised Estimating Equations")
>   } else {
>     out = list(est.method="Maximum Likelihood")
>   }
>   out$sig2 <- x$sig2
>   out$rho  <- x$rho
>   class(out) <- "trim.summary"
>   out
> }
> print.trim.summary <- function(x) {
>   printf("\nEstimation method = %s\n", x$est.method)
>   if (is.finite(x$sig2)) printf("  Estimated Overdispersion     = %f\n", x$sig2)
>   if (is.finite(x$rho))  printf("  Estimated Serial Correlation = %f\n", x$rho)
> }
```

3.3 Coefficients

3.3.1 Extract

```
> coef.trim <- function(x, which=c("additive","multiplicative","both")) {
>   stopifnot(class(x)=="trim")
>   which <- match.arg(which)
```

Last 4 columns contain the additive and multiplicative parameters. Select the appropriate subset from these, and all columns before these 4.

```
>   n = ncol(x$coefficients)
>   stopifnot(n>=4)
>   if (which=="additive") {
>     cols <- 1:(n-2)
>   } else if (which=="multiplicative") {
>     cols <- c(1:(n-4), (n-1):n)
>   } else if (which=="both") {
>     cols = 1:n
>   } else stop(sprintf("Invalid options which=%s", which))
>   out <- x$coefficients[cols]   # ??? does not return correct function output
>   out
> }
```

3.4 Time totals

3.4.1 Extract

```
> totals <- function(x, which=c("imputed","model","both")) {
>   stopifnot(class(x)=="trim")
```

Select output columns from the pre-computed time totals

```
>   which <- match.arg(which)
>   if (which=="model") {
>     totals = x$time.totals[c(1,2,3)]
>   } else if (which=="imputed") {
>     totals = x$time.totals[c(1,4,5)]
>   } else if (which=="both") {
>     totals <- x$time.totals
>   } else stop(sprintf("Invalid options which=%s", which))
>   totals
> }
```

3.5 Reparameterisation of Model 3

3.5.1 extract

```
> linear <- function(x) {
>   stopifnot(class(x)=="trim")
>   stopifnot(x$model==3)
>   structure(list(trend=x$linear.trend, dev=x$deviations), class="trim.linear")
> }
```

3.5.2 print

```
> print.trim.linear <- function(x) {
>   stopifnot(class(x)=="trim.linear")
>   printf("Linear Trend + Deviations for Each Time\n")
>   print(x$trend, row.names=TRUE)
>   printf("\n")
>   print(x$dev, row.names=FALSE)
> }
```

3.6 Plotting

Plotting of data is easier if we convert it to a data table. Here is a function that does the conversion.

```
> mat2df <- function(m, src=NA) {
>   nsite <- nrow(m)
>   ntime <- ncol(m)
>   df <- data.frame(
>     Site  = factor(rep(1:nsite, times=ntime)),
>     Time  = rep(1:ntime, each=nsite),
>     Count = as.vector(m)
>   )
>   df <- df[order(df$Site, df$Time), ]   # Sort by site, then by time
>   if (!is.na(src)) df$Source=src   # set optional data source
>   return(df)
> }
```

```
> plot.trim <- function(x) {
```

Prepare for plotting. First convert the estimations μ to a data frame. Because these estimations are for specific time points, we call it the 'discrete' model.

```
>   Discrete <- rbind(
>     mat2df(x$data,  "Data"),
>     mat2df(x$mu,    "Model")
>   )
>   Discrete <- subset(Discrete, is.finite(Count))
```

Similarly create a data frame for estimations applied to continuous time (the 'continuous' model)

```
>   if (x$model!=3) {
>     ctime <- seq(1, x$ntime, length.out=100)
>     CModel <- data.frame()
>     for (i in 1:x$nsite) {
>       if (x$model==1) {
>         tmp <- data.frame(Site=i, Time=ctime, Count=exp(out$alpha[i]))
>       } else if (out$model==2) {
>         tmp <- data.frame(Site=i, Time=ctime, Count=exp(out$alpha[i] + out$beta*(ctime-1.0)))
>       }
>       CModel <- rbind(CModel, tmp)
>     }
>     CModel$Site <- factor(CModel$Site)
>   }
```

Plot data and model (both discrete and continuous)

```
>   g <- ggplot(Discrete, aes(x=Time, y=Count, colour=Site)) + theme_bw()
>   g <- g + geom_point(aes(shape=Source), size=4)
>   g <- g + scale_shape_manual(values=c(1,20))
>   if (model!=3) g <- g + geom_path(data=CModel)
>   g <- g + scale_x_continuous(breaks=1:x$ntime)
```

```
>   g <- g + labs(shape="")
>   if (nchar(title)>0) g <- g + labs(title=title)
```

Add the overall trend (based on the imputed)

```
>   intercept <- x$overall.slope$imp$coef[[1]][1]
>   slope     <- x$overall.slope$imp$coef[[1]][2]
>   t <- seq(1, x$ntime, length.out=100)
>   y <- exp(intercept + slope*t)
>   trend <- data.frame(Time=t, Count=y, Site=NA)
>   g <- g + geom_path(data=trend)
>   print(g)
> }
```

3.6.1 Plotting

```
> plot_data_df <- function(df, title="") {
>   ntime <- max(df$Time)
```

remove NA rows

```
>   df <- subset(df, is.finite(Count))
>   g <- ggplot(df, aes(x=Time,y=Count,colour=Site)) + theme_bw()
>   g <- g + geom_path(linetype="dashed")
>   g <- g + geom_point(size=4, shape=20)
>   g <- g + scale_x_continuous(breaks=1:ntime)
>   if (nchar(title)) g <- g + labs(title=title)
>   print(g)
> }
> plot_data_mat <- function(m, ...) {
>   df <- mat2df(m)
>   plot_data_df(df, ...)
> }
```

TRIM code documentation Patrick Bogaart September 11, 2016

4 Wald tests

4.1 Theory

TRIM provides a number of tests for the significance of groups of parameters. These so called Wald-tests are based on the estimated covariance matrix of the parameters, and since this covariance matrix takes the overdispersion and serial correlation into account (if specified), these tests are valid not only if the counts are assumed to be independent Poisson observations but also if σ and/or ρ are estimated.

The form of the Wald-statistic for testing simultaneously whether several parameters are different from zero is

$$W = \hat{\theta}^T \left[\mathrm{var}(\hat{\theta})\right]^{-1} \hat{\theta} \tag{23}$$

with $\hat{\theta}$ a vector containing the parameter estimates to be tested, and $\mathrm{var}(\hat{\theta})$ the covariance matrix of $\hat{\theta}$. For the univariate case (i.e., θ is a scalar), Eqn (23) reduces to the univariate case

$$W = \hat{\theta}^2 / \mathrm{var}(\hat{\theta}) \tag{24}$$

The following Wald-tests are performed by TRIM

- Test for the significance of the slope parameter (model 2).

- Tests for the significance of changes in slope (model 2).
- Test for the significance of the deviations from a linear trend (model 3).
- Tests for the significance of the effect of each covariate (models 2 and 3).

The Wald-tests are asymptotically χ^2_{df} distributed, with the number of degrees of freedom equal to the rank of the covariance matrix var($\hat{\theta}$). The hypothesis that the tested parameters are zero is rejected for large values of the test-statistic and small values of the associated significance probabilities (denoted by p), so parameters are significantly different from zero if p is smaller than some chosen significance level (customary choices are 0.01, 0.05 and 0.10).

4.2 Computation

```
> wald <- function(x) UseMethod("wald")

> wald.trim <- function(z)
> {
```

Collect TRIM output variables that we need here.

```
>   model  <- z$model
>   ntime  <- z$ntime
>   nbeta  <- length(z$beta)
>   nbeta0 <- z$nbeta0
>   covars <- z$covars
>   ncovar <- z$ncovar
>   nclass <- z$nclass
>   beta <- z$beta
>   var_beta <- z$var_beta

>   if (model==3 && ncovar==0) {
>     gstar <- z$gstar
>     var_gstar <- z$var_gstar
>   }

>   wald <- list()  # Create empty output object
```

Test for the significance of the slope parameter ———————

This test applies to the case where model 2 is used without covariates or changepoints. There thus is a single β representing the trend for all sites and throughout the whole period, and the univariate approach Eqn (24) applies.

```
>   if (model==2 && nbeta==1 && ncovar==0) {
>     theta <- as.numeric(beta)
>     var_theta <- as.numeric(var_beta)
>     W  <- theta^2 / var_theta  # Compute the Wald statistic by (24)
>     df <- 1  # Degrees of freedom
>     p  <- 1 - pchisq(W, df=df)  # p-value, based on W being χ² distributed.
>     wald$slope <- list(W=W, df=df, p=p)
>   }
```

Tests for the significance of changes in slope ———————

When model 2 is used with changepoints, β is now a vector slope parameters, and the Wald test is used to test if these slopes significantly change after a changepoint. Thus, the Wald test is not applied to individual slope magnitudes β_i, but on the *change in* slope β'_i, where

$$\beta'_i = \beta_i - \beta_{i-1}$$

and

$$\beta'_1 = \beta_1$$

The vector β' can be obtained from the linear tranformation $\beta' = A\beta$ where transformation matrix A is a simple banded matrix structured as

$$A = \begin{pmatrix} 1 & 0 & 0 & \cdots \\ -1 & 1 & 0 & \cdots \\ 0 & -1 & 1 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

that is,

$$A_{i,j} = \begin{cases} 1 & \text{for } i = j, \\ -1 & \text{for } i = j + 1, \\ 0 & \text{otherwise.} \end{cases} \tag{25}$$

```
>   else if (model==2 && nbeta>1 && ncovar==0) {
>     A <- diag(nbeta)  # Start with a diagonal matrix
>     idx <- row(A)==(col(A)+1)  # The band just below the diagonal
>     A[idx] <- -1
>     dbeta <- A %*% beta
```

The covariance matrix of β', $V^{\beta'}$, can be obtained from V^{β} by applying the Taylor (???) delta (???) method

$$V^{\beta'} = AV^{\beta}A^T$$

and the resulting diagonal elements can be taken as the variance of the corresponding β':

$$\text{var}(\beta'_i) = V^{\beta'}_{i,i}$$

```
>     var_dbeta <- A %*% var_beta %*% t(A)
```

Note that the Wald test is applied to each change point individually.

```
>     theta <- as.numeric(dbeta)
>     var_theta <- diag(var_dbeta)
>     W <- theta^2 / var_theta  # Eqn (24)
>     df <- 1  # degrees of freedom
>     p  <- 1 - pchisq(W, df=df)  # p-value, based on W being χ² distributed.
>     wald$dslope <- list(W=W, df=df, p=p)
>   }
```

A variant is the same test for multiple covariates

```
>   else if (model==2 && nbeta>1 && ncovar>0) {
```

Again compute the transformation matrix

```
>     A <-diag(nbeta0)
>     idx <- row(A)==(col(A)+1)  # band just below the diagonal
>     A[idx] <- -1
```

Parameter vector β and it's covariance matrix V^{β} consists of 'blocks' representing either the baseline slopes β_0 or the impact of covariates β_k. If we have n changepoints, then these blocks are $n \times 1$ and $n \times n$ for β and V^{β}, respectively. First create an index for the first block.

```
>     nblock = sum(nclass-1)+1
>     stopifnot(nblock*nbeta0==nbeta)
>     idx0 = 1:nbeta0
```

Again, β' is easily computed from β, except that the transformation $\beta' = A\beta$ is applied to each block

```
>     dbeta = matrix(0, nbeta, 1)
>     for (i in 1:nblock) {
>       idx = idx0 + nbeta0*(i-1)
>       dbeta[idx, ] <- A %*% beta[idx, ]
>     }
```

idem for the covariance matrix

```
>     var_dbeta <- matrix(0, nbeta, nbeta)
>     for (i in 1:nblock) {
>       ridx = seq(to=i*nbeta0, len=nbeta0)  # ((i-1)*nbeta0+1) : (i*nbeta0)
>       for (j in 1:nblock) {
>         cidx <- seq(to=j*nbeta0, len=nbeta0)
>         var_dbeta[ridx,cidx] <- A %*% var_beta[ridx,cidx] %*% t(A)
>       }
>     }
```

Compute a Wald statistic for each changepoint

```
>     W = numeric(nbeta0)
>     for (b in 1:nbeta0) {
>       idx <- seq(from=b, by=nbeta0, len=nblock)
>       theta = dbeta[idx]
>       var_theta = var_dbeta[idx,idx]
>       W[b] = t(theta) %*% solve(var_theta) %*% theta
>     }
>     df <- nblock
>     p <- 1 - pchisq(W, df)
>     wald$dslope <- list(W=W, df=df, p=p)
>   }
```

Test for the significance of the deviations from a linear trend ————

For Model 3, we use the Wald test to test if the residuals around the overall trend (i.e., the γ_j^*) significantly differ from 0. For this case, the vectorized Wald equation (23) is used. Note that this test is only performed when there are no covariates.

```
>   else if (model==3 && ncovar==0) {
>     J <- ntime
>     theta <- matrix(gstar)  # Column vector of all J γ*.
>     var_theta <- var_gstar[-1,-1]  # Covariance matrix; drop the β* terms.
```

We now have J equations, but due to the double contraints 2 of them are linear dependent on the others. Let's confirm this:

```
>     eig <- eigen(var_theta)$values
>     stopifnot(sum(eig<1e-7)==2)
```

Shrink θ and it's covariance matrix to remove the dependent equations.

```
>     theta <- theta[3:J]
>     var_theta <- var_theta[3:J, 3:J]
```

```
>     W <- t(theta) %*% solve(var_theta) %*% theta  # Eqn (23)
>     W <- as.numeric(W)  # Convert from 1 × 1 matrix to proper atomic
>     df <- J-2  # degrees of freedom
>     p  <- 1 - pchisq(W, df=df)  # p-value, based on W being χ² distributed.
```

```
>     wald$deviations <- list(W=W, df=df, p=p)
>   }
>   else if (model==3 && ncovar>0) {
```

pass

```
>   } else stop("Can't happen")
```

Tests for the significance of the effect of each covariate ————————

As explained in ????, for both models 2 and 3, the covariate effects are modelled as additions β_k to the baseline slope parameters β_0 which represent the first class of all covariates.

```
>   if (ncovar>0) {
>     wald$covar <- data.frame(Covariate=names(covars), W=0, df=0, p=0)
```

```
>     size <- (nclass-1)*nbeta0  # size of covariate block witin total β vector
>     last <- cumsum(size)+nbeta0  # last element of covariate block
>     first <- last - size + 1  # first element
>     for (i in 1:length(covars)) {
>       idx <- first[i] : last[i]
>       theta <- matrix(beta[idx])
>       var_theta <- var_beta[idx, idx]
>       W = t(theta) %*% solve(var_theta) %*% theta
>       wald$covar$W[i]     <- W
>       wald$covar$df[i]    <- nbeta0
>       wald$covar$p[i]     <- 1 - pchisq(W, df=nbeta0)
>     }
>   }
```

Output results in a list with type 'trim.wald' to enable specialized further processing.

```
>   class(wald) <- "trim.wald"
>   wald
> }
```

4.3 Printing

A simple printing function is provided that mimics the output of TRIM for Windows.

```
> print.trim.wald <- function(w) {
>   stopifnot(class(w)=="trim.wald")

>   if (!is.null(w$covar)) {
>     printf("Wald test for significance of covariates\n")
>     print(w$covar, row.names=FALSE)
>     printf("\n")
>   }

>   if (!is.null(w$slope)) {
>     printf("Wald test for significance of slope parameter\n")
>     printf("  Wald = %.2f, df=%d, p=%f\n", w$slope$W, w$slope$df, w$slope$p)
>   } else if (!is.null(w$dslope)) {
>     printf("Wald test for significance of changes in slope\n")
>     df = data.frame(Changepoint = 1:length(w$dslope$W),
>                     Wald_test = w$dslope$W, df = w$dslope$df, p = w$dslope$p)
>     print(df, row.names=FALSE)
>   } else if (!is.null(w$deviations)) {
>     printf("Wald test for significance of deviations from linear trend\n")
>     printf("  Wald = %.2f, df=%d, p=%f\n", w$deviations$W, w$deviations$df, w$deviations$p)
>   }
> }
```

5 Indices

5.1 Internal workhorse function

```
> .index <- function(tt, var_tt, b) {
```

Time index τ_j is defined as time totals μ_+, normalized by the time total for the base year, i.e.

$$\tau_j = \mu_{+j}/\mu_{+b}$$

where $b \in [1 \ldots J]$ indicates the base year.

```
>   tau <- tt / tt[b]
```

Uncertainty is again quantified as a standard error $\sqrt{\text{var}(\cdot)}$, approximated using the delta method, now extended for the multivariate case:

$$\text{var}(\tau_j) = \text{var}(f(\mu_{+_b}, \mu_{+_j})) = d^T V(\mu_{+_b}, \mu_{+_j}) d \tag{26}$$

where d is a vector containing the partial derivatives of $f(\mu_{+_b}, \mu_{+_j})$

$$d = \begin{pmatrix} -\mu_{+_j} \mu_{+_b}^{-2} \\ \mu_{+_b}^{-1} \end{pmatrix} \tag{27}$$

and V the covariance matrix of μ_{+_b} and μ_{+_j}:

$$V(\mu_{+_b}, \mu_{+_j}) = \begin{pmatrix} \text{var}(\mu_{+_b}) & \text{cov}(\mu_{+_b}, \mu_{+_j}) \\ \text{cov}(\mu_{+_b}, \mu_{+_j}) & \text{var}(\mu_{+_j}) \end{pmatrix} \tag{28}$$

Note that for the base year b, where $\tau_b \equiv 1$, Eqn (26) results in $\text{var}(\tau_b) = 0$, which is also expected conceptually because τ_b is not an estimate but an exact and fixed result.

```
>   J <- length(tt)
>   var_tau <- numeric(J)
>   for (j in 1:J) {
>     d <- matrix(c(-tt[j] / tt[b]^2, 1/tt[b]))
>     V <- var_tt[c(b,j), c(b,j)]
>     var_tau[j] <- t(d) %*% V %*% d
>   }
>   out = list(tau=tau, var_tau=var_tau)
> }
```

5.2 User interface

```
> index <- function(trm, base=1, which=c("imputed","model","both")) {
>   stopifnot(class(trm)=="trim")
```

Computation and output is user-configurable

```
>   which <- match.arg(which)
>   if (which=="model") {
```

Call workhorse function to do the actual computation

```
>     mod <- .index(trm$tt_mod, trm$var_tt_mod, base)
```

Store results in a data frame

```
>     out = data.frame(time  = 1:trm$ntime,
>                      model = mod$tau,
>                      se_mod = sqrt(mod$var_tau))
>   } else if (which=="imputed") {
```

Idem, using the imputed time totals instead

```
>     imp <- .index(trm$tt_imp, trm$var_tt_imp, base)
>     out = data.frame(time    = 1:trm$ntime,
>                      imputed = imp$tau,
>                      se_imp  = sqrt(imp$var_tau))
>   } else if (which=="both") {
```

Idem, using both modelled and imputed time totals.

```
>     mod <- .index(trm$tt_mod, trm$var_tt_mod, base)
>     imp <- .index(trm$tt_imp, trm$var_tt_imp, base)
>     out = data.frame(time    = 1:trm$ntime,
>                      model   = mod$tau,
```

```
>                       se_mod  = sqrt(mod$var_tau),
>                       imputed = imp$tau,
>                       se_imp  = sqrt(imp$var_tau))
>   } else stop("Can't happen")  # because other cases are catched by match.arg()
>   out
> }
```

6 Overall slope

```
> overall <- function(x, which=c("imputed","model")) {
>   stopifnot(class(x)=="trim")
>   which = match.arg(which)
```

extract vars from TRIM output

```
>   tt_mod <- z$tt_mod
>   tt_imp <- z$tt_imp
>   var_tt_mod <- z$var_tt_mod
>   var_tt_imp <- z$var_tt_imp
>   ntime <- z$ntime
```

The overall slope is computed for both the modeled and the imputed μ_+'s. So we define a function to do the actual work

```
>   .compute.overall.slope <- function(tt, var_tt) {
>     stopifnot(length(tt)==ntime)
>     J <- ntime
```

Use Ordinary Least Squares (OLS) to estimate slope parameter β

```
>     X <- cbind(1, seq_len(ntime))  # design matrix
>     y <- matrix(log(tt))
>     bhat <- solve(t(X) %*% X) %*% t(X) %*% y  # OLS estimate of b = (α, β)^T
>     yhat <- X %*% bhat
```

Apply the sandwich method to take heteroskedasticity into account

```
>     dvtt <- 1/tt_mod  # derivative of log μ+
>     Om <- diag(dvtt) %*% var_tt %*% diag(dvtt)  # var(log μ+)
>     var_beta <- solve(t(X) %*% X) %*% t(X) %*% Om %*% X %*% solve(t(X) %*% X)
>     b_err <- sqrt(diag(var_beta))
```

Compute the p-value, using the t-distribution

```
>     df <- ntime - 2
>     t_val <- bhat[2] / b_err[2]
>     p <- 2 * pt(abs(t_val), df, lower.tail=FALSE)
```

Also compute effect size as relative change during the monitoring period.

```
>     effect <- abs(yhat[J] - yhat[1]) / yhat[1]
```

Reverse-engineer the SSR (sum of squared residuals) from the standard error

```
>     j <- 1:J
>     D <- sum((j-mean(j))^2)
>     SSR <- b_err[2]^2 * D * (J-2)
```

Export the results

```
>     df <- data.frame(
>       Additive       = bhat,
>       std.err.       = b_err,
>       Multiplicative = exp(bhat),
```

```
>       std.err.       = exp(bhat) * b_err,
>       row.names      = c("Intercept","Slope"),
>       check.names    = FALSE
>     )
>     list(coef=df,p=p, effect=effect, J=J, tt=tt, err=z$time.totals[[3]], SSR=SSR)
>   }
>   if (which=="imputed") {
>     out = .compute.overall.slope(tt_imp, var_tt_imp)
>     out$src = "imputed"
>   } else if (which=="model") {
>     out = .compute.overall.slope(tt_mod, var_tt_mod)
>     out$src = "model"
>   } else stop("Can't happen")
>   structure(out, class="trim.overall")
> }
```

6.0.1 Extract

6.0.2 Print

```
> print.trim.overall <- function(x) {
>   stopifnot(class(x)=="trim.overall")
```

Compute 95% confidence interval of multiplicative slope

```
>   bhat <- x$coef[[3]][2]   # multiplicative trend (i.e., not log-transformed)
>   berr <- x$coef[[4]][2]   # corresponding standard error
>   alpha <- 0.05
>   df <- x$J-2
>   tval <- qt((1-alpha/2), df)
>   blo <- bhat - tval * berr
>   bhi <- bhat + tval * berr
```

Compute effect size

```
>   change <- bhat ^ (x$J-1) - 1
```

Build an informative string

```
>   info <- sprintf("p=%f, conf.int (mul)=[%.f, %f], change=%.2f%%", x$p, blo, bhi, 100*change)
>   printf("Overall slope (%s): %s\n", x$src, info)
>   print(x$coef, row.names=TRUE)
> }
```

6.0.3 Plot

```
> plot.trim.overall <- function(X, imputed=TRUE, ...) {
>   title <- attr(X, "title")
>   J <- X$J
```

Collect all data for plotting: time-totals

```
>   j <- 1:J
>   ydata <- X$tt
```

error bars

```
>   y0 = ydata - X$err
>   y1 = ydata + X$err
```

Trend line

```
>   a <- X$coef[[1]][1]  # intercept
>   b <- X$coef[[1]][2]  # slope
>   x <- seq(1, J, length.out=100)
>   ytrend <- exp(a + b*x)
```

Confidence band

```
>   xconf <- c(x, rev(x))
>   alpha <- 0.05
>   df <- J - 2
>   t <- qt((1-alpha/2), df)
>   dx2 <- (x-mean(j))^2
>   sumdj2 <- sum((j-mean(j))^2)
>   dy <- t * sqrt((X$SSR/(J-2))*(1/J + dx2/sumdj2))
>   ylo <- exp(a + b*x - dy)
>   yhi <- exp(a + b*x + dy)
>   yconf <- c(ylo, rev(yhi))
```

Compute the total range of all plot elements

```
>   xrange = range(x)
>   yrange = range(range(yconf), range(y0), range(y1))
```

Now plot layer-by-layer

```
>   cbred <- rgb(228,26,28, maxColorValue = 255)
>   cbblue <- rgb(55,126,184, maxColorValue = 255)
>   plot(xrange, yrange, type='n', xlab="Time point", ylab="Count", main=title)
>   polygon(xconf, yconf, col=gray(0.9), lty=0)
>   lines(x, ytrend, col=cbred, lwd=3)
>   segments(j,y0, j,y1, lwd=3, col=gray(0.5))
>   points(j, ydata, col=cbblue, type='b', pch=16, lwd=3)
> }
```