1 Introduction

This document describes the core TRIM function.

Define a convenience function for console output

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

1.1 Interface

```
> #' TRIM workhorse function
> #'
> #' @param count a numerical vector of count data.
> #' @param time.id a numerical vector time points for each count data point.
> #' @param site.id a numerical vector time points for each count data point.
> #' @param covars an optional list of covariates
> #' @param model a model type selector
> #' @param serialcor a flag indication of autoorrelation has to be taken into account.
> #' @param overdisp a flag indicating of overdispersion has to be taken into account.
> #' @param changepoints a numerical vector change points (only for Model 2)
> #'
> #' @return a list of class \code{trim}, that contains all output, statistiscs, etc.
> #'   Usually this information is retrieved by a set of postprocessing functions
> #' @export
> #'
> #' @examples
> #' z <- trim_estimate(...)
> trim_estimate <- function(count, time.id, site.id, covars=NA,
>                           model=c(1,2,3), serialcor=FALSE, overdisp=FALSE,
>                           changepoints=1L) {
```

1.2 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)
```

`model` should be in the range 1 to 3

```
>   stopifnot(model %in% 1:3)
```

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)
>   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
```

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) {
>     stopifnot(all(changepoints>=1L))
>     stopifnot(all(changepoints<ntime))
>     stopifnot(all(diff(changepoints)>0))
>     if (changepoints[1]!=1L) changepoints = c(1L, changepoints)
>   }
```

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 column3fff
>   }
```

1.2.1 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

```
>     beta = matrix(0, length(changepoints), 1)
>   } else if (model==3) {
```

For model 3, we have one β per time $j > 1$; all are initialized to 0 (no time effects)

```
>     beta <- matrix(0, ntime-1,1)  # 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.

1.3.1 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 1.3.3).

```
>   update_alpha <- function(method=c("ML","GEE")) {
>     for (i in 1:nsite) {
>       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))
>     }
>   }
```

1.3.2 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 1.3.6) and U_b^* is a Fisher Scoring matrix (see Section 1.3.6). Note that the 'improvement' as defined by (1.3.2) can actually results in a decrease in model fit. These

cases are identified by measuring the model Likelihood Ratio (Eqn (12)). 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
>   }
```

1.3.3 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 1.3.4) 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
>   }
```

1.3.4 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 1.3.5)

```
>   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
>   }
```

1.3.5 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])
>   }
```

1.3.6 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) {
>       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)
>     }
>   }
```

1.3.7 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) {
>       mu[i, ] <<- exp(alpha[i] + B %*% beta)
>     }
```

clear estimates for non-observed cases, if required.

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

1.3.8 Likelihood

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

1.3.9 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.

```
>   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
>       printf(" 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
>   }
```

1.3.10 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) {
>     printf("Iteration %d (%s)", iter, method)

>     update_alpha(method)
>     update_mu(fill=FALSE)
>     if (method=="GEE") {
>       update_r()
>       update_sig2()
```

```
>       update_rho()
>       update_R()
>     }
>     update_V(method)
>     subiters <- update_beta(method)
>     printf(", %d subiters", subiters)
>     printf(", lik=%.3f", likelihood())
>     if (overdisp)  printf(", sig^2=%.5f", sig2)
>     if (serialcor) printf(", rho=%.5f;", rho)
>     convergence <- check_convergence(iter)
>     if (convergence && method==final_method) {
>       printf("\nConvergence reached\n")
>       break
>     } else if (convergence) {
>       printf("\nChanging ML --> GEE\n")
>       method = "GEE"
>     } else {
>       printf("\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)
```

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, data=f, nsite=nsite, ntime=ntime,
>             model=model, mu=mu, imputed=imputed, alpha=alpha, beta=beta)
>   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) {
>     beta     <- as.vector(beta)
>     var_beta <- -solve(i_b)
>     se_beta  <- sqrt(diag(var_beta))
```

Again, results are stored in the TRIM object

```
>     z$coefficients <- data.frame(
>       Additive      = beta,
>       std.err.      = se_beta,
>       Mutiplicative = exp(beta),
>       std.err.      = exp(beta) * se_beta,
>       check.names   = FALSE  # to allow for 2 "std.err." columns
>     )
>     printf("----\n")
>     str(z$coefficients)
>     printf("----\n")
>     row.names(z$coefficients) <- "Slope"
>   }
>   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 $g_j = e^{\gamma_j}$.

```
>     gamma     <-  matrix(c(0, as.vector(beta)))  # Add γ1 ≡ 1, and cast as column vector
>     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)
```

Because $\gamma_1 \equiv 1$, 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' rows and columns.

```
>     var_gamma <- cbind(0, rbind(0, var_gamma))
```

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
```

Again, results are stored in the TRIM object

```
>     z$coefficients <- data.frame(
>       Time          = 1:ntime,
>       Additive      = gamma,
>       std.err.      = se_gamma,
>       Mutiplicative = g,
>       std.err.      = g * se_gamma,
>       check.names   = FALSE  # to allow for 2 "std.err." columns
>     )
>   }
```

1.5.3 Goodness-of-fit

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

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

$$\chi^2 = \sum_{ij} \frac{f_{i,j} - \mu_{i,j}}{\mu_{i,j}} \tag{11}$$

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)
```

Results are stored in the TRIM output object.

```
>   z$chi2 <- list(chi2=chi2, df=df, p=p)
```

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

$$\mathrm{LR} = 2 \sum_{ij} f_{ij} \log \frac{f_{i,j}}{\mu_{i,j}} \tag{12}$$

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)
>   z$LR <- list(LR=LR, df=df, p=p)
```

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

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

1.5.4 Time Totals

Recompute i_b with final μ's

```
>   ib <- 0
>   for (i in 1:nsite) {
>     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) {
>     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) {
>     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_tau_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) {
>     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_tau_obs_old <- GddG + GFminH %*% solve(E) %*% t(GFminH)
```

Now compute the variance due to observations

```
>   var_tau_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_tau_obs_new = var_tau_obs_new + V
>   }
```

Combine

```
>   var_tau_imp = var_tau_mod - var_tau_obs_old + var_tau_obs_new
```

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

```
>   tau_mod    <- colSums(mu)
>   se_tau_mod <- round(sqrt(diag(var_tau_mod)))
>   tau_imp     <- colSums(imputed)
>   se_tau_imp <- round(sqrt(diag(var_tau_imp)))
>   z$time.totals <- data.frame(
>     Time       = 1:ntime,
>     Model      = round(tau_mod),
>     std.err.   = se_tau_mod,
>     Imputed    = round(tau_imp),
>     std.err.   = se_tau_imp,
>     check.names = FALSE
>   )
```

1.5.5 Time indices

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

$$\tau_j = \mu_{+j}/\mu_{+1}$$

. Indices are computed for both the modelled and the imputed counts.

```
>   ti_mod <- tau_mod / tau_mod[1]
>   ti_imp <- tau_imp / tau_imp[1]
```

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

$$\text{var}(\tau_j) = \text{var}(f(\mu_{+1}, \mu_{+j})) = d^T V(\mu_{+1}, \mu_{+j}) d \tag{13}$$

where d is a vector containing the partial derivatives of $f(\mu_{+1}, \mu_{+j})$

$$d = \begin{pmatrix} -\mu_{+j}\mu_{+1}^{-2} \\ \mu_{+1}^{-1} \end{pmatrix} \tag{14}$$

and V the covariance matrix of μ_{+1} and μ_{+j}:

$$V(\mu_{+1}, \mu_{+j}) = \begin{pmatrix} \text{var}(\mu_{+1}) & \text{cov}(\mu_{+1}, \mu_{+j}) \\ \text{cov}(\mu_{+1}, \mu_{+j}) & \text{var}(\mu_{+j}) \end{pmatrix} \tag{15}$$

Note that for the base year, where $\tau_1 \equiv 1$, Eqn (13) results in $\text{var}(\tau_1) = 0$, which is also expected conceptually because τ_1 is not an estimate but an exact and fixed result.

```
>   var_ti_mod <- numeric(ntime)
>   for (j in 1:ntime) {
>     d <- matrix(c(-tau_mod[j] / tau_mod[1]^2, 1/tau_mod[1]))
>     V <- var_tau_mod[c(1,j), c(1,j)]
>     var_ti_mod[j] <- t(d) %*% V %*% d
>   }
>   se_ti_mod <- sqrt(var_ti_mod)
```

Similarly for the Indices based on the imputed counts

```
>   se_ti_imp <- numeric(ntime)
>   for (j in 1:ntime) {
>     d <- matrix(c(-tau_imp[j]/tau_imp[1]^2, 1/tau_imp[1]))
>     V <- var_tau_imp[c(1,j), c(1,j)]
>     se_ti_imp[j] <- sqrt(t(d) %*% V %*% d)
>   }
```

Store in TRIM output object

```
>   z$time.index <- data.frame(
>     Time     = 1:ntime,
>     Model    = ti_mod,
>     std.err. = se_ti_mod,
>     Imputed  = ti_imp,
>     std.err. = se_ti_imp,
>     check.names = FALSE
>   )
```

1.5.6 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

$$\log \mu_{ij} = \alpha_i + \gamma_j, \tag{16}$$

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

$$\log \mu_{ij} = \alpha_i^* + \beta^* d_j + \gamma_j^*, \tag{17}$$

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.

$$\sum_j \gamma_j^* = 0 \quad \text{and} \quad \sum_j d_j \gamma_j^* = 0. \tag{18}$$

Using these constraints we obtain the equations:

$$\log \mu_{ij} = \alpha_i^* + \beta^* d_j + \gamma_j^* = \alpha_i + \gamma_j \tag{19}$$

$$\sum_j \log \mu_{ij} = J\alpha_j^* = J\alpha_i + \sum_j \gamma_j \tag{20}$$

$$\sum_j d_j \log \mu_{ij} = \beta^* \sum_j d_j^2 = \sum_j d_j \gamma_j, \tag{21}$$

where (19) is the re-parameterization equation itself and (20) and (21) are obtained by using the constraints (18)

From (20) we have that $\alpha_i^* = \alpha_i + \frac{1}{J}\sum_j \gamma_j$. Now, by using the equations (19) thru (21) and defining $D = \sum_j d_j^2$, we can express the parameters β^* and γ^* as functions of the parameters γ as follows:

$$\beta^* = \frac{1}{D}\sum_j d_j\gamma_j, \tag{22}$$

$$\begin{aligned}\gamma_j^* &= \alpha_i + \gamma_j - \alpha_i^* - \beta^* d_j \quad \text{(using (5))} \\ &= \alpha_i - \left(\alpha_i + \frac{1}{J}\sum_j \gamma_j\right) + \gamma_j - d_j\frac{1}{D}\sum_j d_j\gamma_j \\ &= \gamma_j - \frac{1}{J}\sum_j \gamma_j - d_j\frac{1}{D}\sum_j d_j\gamma_j. \end{aligned} \tag{23}$$

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

$$\begin{pmatrix} \beta^* \\ \boldsymbol{\gamma}^* \end{pmatrix} = \mathbf{T}\boldsymbol{\gamma}, \tag{24}$$

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 (22) and (23) 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) {
>     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 %*% gamma
>     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

$$V\begin{pmatrix} \beta^* \\ \boldsymbol{\gamma}^* \end{pmatrix} = \mathbf{T}V(\boldsymbol{\gamma})\mathbf{T}^T \tag{25}$$

```
>     var_gstar <- TT %*% var_gamma %*% t(TT)
>     se_bstar  <- sqrt(diag(var_gstar))[1]
>     se_gstar  <- sqrt(diag(var_gstar))[2:(ntime+1)]
>     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
>     )
>   }
```

1.5.7 Wald test

```
>   if (model==2) {
>     theta = beta[1]
>     var_theta = var_beta[1,1]
>     W <- t(theta) %*% solve(var_theta) %*% theta  # Compute the Wald statistic
>     W <- as.numeric(W)  # Convert from 1 × 1 matrix to proper atomic
>     df <- 1  # degrees of freedom
>     p  <- 1 - pchisq(W, df=df)  # p-value, based on W being χ² distributed.
>     z$wald <- list(model=model, W=W, df=df, p=p)}
```

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. The Wald statistic used for this is defined as

$$W = \theta^T \left(\mathrm{var}(\theta)\right)^{-1} \theta \tag{26}$$

```
>   if (model==3) {
>     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  # Compute the Wald statistic
>     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.
>     z$wald <- list(model=model, W=W, df=df, p=p)
>   }
```

1.5.8 Overall slope

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) {
```

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/tau_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)
>   }
```

Compute the overall trends for both the modelled and the imputed counts, and store the results in the TRIM output

```
>   z$overall <- list()
>   z$overall$mod  <- .compute.overall.slope(tau_mod, var_tau_mod)
>   z$overall$imp  <- .compute.overall.slope(tau_imp, var_tau_imp)
```

1.6 Return results

The TRIM result is returned to the user...

```
>   z
> }
```

...which ends the main TRIM function.