countimp 1.0 – A Multiple Imputation Package for Incomplete Count Data

Kristian Kleinke[*] and Jost Reinecke
University of Bielefeld, Faculty of Sociology

Version 1.0 – 2013-05-15

Abstract

Special data types like count data require special analysis and imputation techniques. Yet, currently available multiple imputation tools are very limited with regard to count data. The **countimp** package provides easy to use multiple imputation (MI) procedures for incomplete count data based on either a Bayesian regression approach (Rubin, 1987) or on a bootstrap regression approach within a chained equations MI framework (van Buuren, Brand, Groothuis-Oudshoorn, & Rubin, 2006; van Buuren & Groothuis-Oudshoorn, 2011). Our software works as an add-on for the popular and powerful **mice** package in R (van Buuren & Groothuis-Oudshoorn, 2011). The current version of **countimp** supports ordinary count data imputation under the Poisson model, imputation of incomplete overdispersed count data under either the Quasi-Poisson or the Negative Binomial model, imputation of zero-inflated ordinary or overdispersed count data based on a zero-inflated Poisson or Negative Binomial model, and imputation of multilevel count data based on a generalized linear mixed effects model (overdispersion and zero-inflation are supported).

Keywords: *missing data, count data*

[*]kristian.kleinke@uni-bielefeld.de

1 Introduction and overview

Count data require special analysis techniques: Ordinary count data are typically analyzed using some kind of Poisson model. Overdispersed count data, i.e., count data that have a larger variance in comparison to the mean are usually modeled by Quasi-Poisson or Negative Binomial (NB) models. Zero-inflated count data, i.e., count data that exhibit an excess number of zero counts can be analyzed by either zero-inflated Poisson or NB models, or by hurdle models (Zeileis, Kleiber, & Jackman, 2008). Furthermore, generalized linear mixed effects Poisson or NB models can for example be used to analyze multilevel count data.

Missing data methods and especially multiple imputation procedures for count data however are very sparse. In practice, the missing data problem in count variables is typically handled by (a) ignoring that the data are counts and by proceeding as if they were continuous, (b) by treating the data as categorical or ordinal variables and using imputation techniques for these data like polytomous regression, or (c) by applying some normalizing transformation like a square-root transformation and using imputation techniques for normal data (cf. Landerman, Land, & Pieper, 1997). We regard these solutions as quick fixes (that may work in some settings), but not as an optimal and general solution to adequately analyze incomplete count data and get precise and unbiased parameter estimates as well as standard errors in a wide variety of scenarios. Erdman, Jackson, & Sinko (2008) have demonstrated that analyzing count data with suboptimal models yields biased results. We think that what is true for data analysis will also be true for data imputation. Imputation procedures should therefore be tailored to the problem at hand (e.g. overdispersed, zero-inflated, or multilevel count data). Unfortunately, existing MI solutions for count data support only rather basic models:

The R package **mi** for example offers multiple imputation routines for incomplete ordinary and overdispersed count data using Bayesian Quasi-Poisson regression (Su, Gelman, Hill, & Yajima, 2009). Zero-inflation or multilevel count data are not supported by that package. **IVEware**, which is available as a SAS add-on or standalone version offers count data support, but fits "only" an ordinary Poisson model. **ice** for STATA 12 supports count data imputation under a Poisson or Negative Binomial regression model. However, zero-inflation and multilevel models are not supported. M*plus* version 7 offers full information maximum likelihood (FIML) estimation of incomplete count data models, but multiple imputation routines in M*plus* currently do not support count data.

This lack of general count data support was not very satisfactory and we wanted to create a MI package that can handle all kinds of count data: ordinary, overdispersed, zero-inflated and multilevel count data. Our aim was to build a package that is flexible and easy to use. The imputation procedures should be applicable under missing completely at random and missing at random mechanisms (Rubin, 1976).

Our software uses the multiple imputation by chained equations (MICE) approach to create the imputations (van Buuren et al., 2006; van Buuren & Groothuis-Oudshoorn, 2011) and works as an add-on for the **mice** package in R (van Buuren & Groothuis-Oudshoorn, 2011). The underlying statistical theory stems from Rubin (1987): Our multiple imputation algorithms for count data may be seen as an extension or generalization of Rubin's (1987) Bayesian logistic regression imputation approach to other data types. As an alternative to Bayesian regression, which may not be the best choice in some scenarios – as it assumes a normal distribution of parameters, we offer a bootstrap version of each algorithm. The package is available from `http://www.uni-bielefeld.de/soz/kds/software/countimp_1.0.tar.gz`.

The paper is structured as follows: We begin by introducing the general idea of multiple imputation and the chained equations approach of multiple imputation. We then elaborate on different count data models and describe our missing data algorithms for count data in detail. We demonstrate, how our functions work together with **mice** and give some examples. We finally discuss advantages and limitations of our solutions and outline fruitful avenues for future software development.

2 Theoretical background

2.1 Multiple imputation in a nutshell

Throughout the last two decades, multiple imputation (Rubin, 1987; Schafer, 1997) has become more and more popular and has become one of the standard procedures to handle missing data (Schafer & Graham, 2002).

Multiple imputation procedures are generally preferable to single imputation, as single imputation procedures typically have a problem to produce unbiased standard error erstimates (Schafer & Graham, 2002). Multiple Imputation (MI) means that each missing value is filled in more than once. The resulting *m* complete data sets are then analyzed separately and the *m* statistical results are combined into an overall result using Rubin's rules for MI inference (Rubin, 1987). These rules take variation within and between the *m* completed data sets into account to compute an overall variance estimate. This additional variation is supposed to reflect estimation uncertainty due to missing data in an adequate way. The combined point estimate of the parameter of interest is simply the mean of the *m* parameter estimates, which typically produces a precise estimate of the respective population parameter (Schafer & Graham, 2002).

Various multiple imputation tools for different data types have been proposed, e.g., **norm** for multivariate normal data, **cat** for categorical data, **mix** for data sets containing both continuous and categorical data, and **pan** for panel data (Schafer, 1997; Schafer & Yucel, 2002). Schafer's MI approach – based on theoretical work by Rubin (1987) – is often referred to as the joint modeling approach to multiple imputation (e.g. van Buuren & Groothuis-Oudshoorn, 2011), as his algorithms assume a joint probability distribution of all the variables in the model. A different MI approach – the MICE approach (multiple imputation by chained equations) – imputes missing data separately and iteratively on a variable to variable basis, using a chain of regression models (Raghunathan, Lepkowski, van Hoewyk, & Solenberger, 2001; van Buuren et al., 2006; van Buuren & Groothuis-Oudshoorn, 2011). Advantages and disadvantages of the respective MI frameworks have been reviewed in Kleinke, Stemmler, Reinecke, & Lösel (2011).

How exactly imputations are generated depends on the respective multiple imputation framework: Whereas joint modeling makes it necessary to specify a joint probability distribution of all the variables in the model and draws imputation from the predictive distribution of missing data given the observed data under this joint model, the chained equation approach models each incompletely observed variable separately. For each variable with missing data, a subset of variables in the data set is defined to predict missing data in that variable using some kind of regression approach. Imputations are generated from $\mathsf{P}(Y_j|Y_s,\theta_j)$, where Y_j is the variable containing missing data, Y_s is the subset of variables that is used to model Y_j and θ_j are the parameters to be estimated. In detail, imputations are generated by an iterative procedure: First, the posterior distribution of θ is calculated given the observed data. Then new parameters θ^* are simulated from $\mathsf{P}(\theta|Y_{obs})$, where Y_{obs} denotes the observed part of the data in data set Y. Finally, imputations Y^* are drawn from $\mathsf{P}(Y_{mis}|Y_{obs},\theta^*)$. Steps two and three are repeated m times to obtain the m imputations. From a mathematical and computational standpoint, the only problem is to determine the respective distributions to draw from. In **mice** this is done by iteratively sampling from the conditional distributions using a Gibbs sampler (see van Buuren & Groothuis-Oudshoorn, 2011, for details). Both MI approaches assume that missing data are either missing completely at random (MCAR) or missing at random (MAR) in the sense of Rubin (1976).

We use the chained equations MI framework as basis for our algorithms. We prefer this approach over joint modeling because it is more flexible with regard to "mixed" data sets: It is far easier to impute data sets with variables of various data types like continuous, categorical, and count variables, as one does not have to find a joint model, which in fact may not even exist.

2.2 The mice package in R

The **mice** package in R has a couple of advantages we want to point out and which make the software a good basis for our algorithms: **mice**, like the R environment for statistical computing, is completely open source and available free of charge to all practitioners from `www.r-project.org`. As the source code is completely available, the computation process is transparent. Everybody can see, how the program works and what the program does to produce the results. Furthermore, **mice** already comes with a great variety of imputation solutions and additionally allows to call user-written imputation functions (such as ours). It is thus very flexible and easily extendable. Beyond that, **mice** offers highly helpful tools to assess imputation convergence and quality. Features like rounding, passive imputation and post processing are helpful to ensure that imputations are plausible and within an allowed range. For an overview of all of **mice**'s capabilities and how to use the program, see van Buuren & Groothuis-Oudshoorn (2011). Here, we review only the most basic essentials that are necessary to know in order to create multiple imputations with **mice** and our add-on functions.

Multiple imputations are created by calling `mice()`. The function uses the following arguments:

```
mice(data, m = 5,
    method = vector("character",length=ncol(data)),
    predictorMatrix = (1 - diag(1, ncol(data))),
    visitSequence = (1:ncol(data))[apply(is.na(data),2,any)],
    post = vector("character", length = ncol(data)),
    defaultMethod = c("pmm","logreg","polyreg"),
    maxit = 5,
    diagnostics = TRUE,
    printFlag = TRUE,
    seed = NA,
    imputationMethod = NULL,
    defaultImputationMethod = NULL
    )
```

`m` sets the number of imputations (default is `m = 5`), `maxit` sets the maximum number of iterations for the Gibbs sampler. The imputation procedure with which missing data in a certain variable are imputed is specified via the `method` argument: A not exhaustive overview of imputation methods that are already implemented in MICE is given in Table 1. The procedures are all described in detail in van Buuren & Groothuis-Oudshoorn (2011).

`method` must be a character vector of length equal to the number of variables in the data set. The i^{th} entry in method corresponds to the i^{th} variable in the data set. Completely observed variables have method `""`, indicating that they need not be imputed. The command `imp <- mice( data, method = c( "", "norm", "pmm", "logreg" ) )` would multiply impute the data stored in the object `data` and store the results in an object called `imp`. The first variable in that data set would not be imputed, the second one would be imputed using Bayesian linear

Table 1: Overview of imputation procedures in **mice**

Name	Description	Scale
pmm	predictive mean matching	numeric
norm	Bayesian linear regression	numeric
2l.norm	2-level linear mixed effects model	numeric
logreg	Bayesian logistic regression	factor (2 levels)
polyreg	polytomous regression	factor ($>$ 2 levels)
sample	random sample from observed data	any

regression (`norm`), the third would be filled in by predictive mean matching (`pmm`), and the last one by Bayesian logistic regression (`logreg`). The respective imputation functions are stored internally under the name `mice.impute.name` where `name` identifies the respective imputation function. Thus specifying `"logreg"` as imputation method for a variable internally calls the function `mice.impute.logreg()`. This is important to know when programming self-written imputation procedures. That these can be called by `mice()`, they have to be called `mice.impute.name`, where the `name` part can be any combination of characters. One of our count data imputation functions for example is called `mice.impute.2l.nb2()`. These function can be used for multiple imputing overdispersed two-level count data on the basis of a Negative Binomial model by setting the respective entry in the `method` vector to `"2l.nb2"`.

Selecting the subsets of predictors for each incompletely observed variable is done via the `predictorMatrix` argument. `predictorMatrix` must be a rectangular matrix of dimensions equal to the number of variables in the data set. An example is shown in Table 2. Each row i in that matrix denotes the imputation model of variable V_i. The zeros and ones indicate (0 = no; 1 = yes), if the respective variable V_j is used to predict missing data in V_i. Using the information from the `predictorMatrix`, `mice` automatically creates three objects that are passed on to the respective `mice.impute.name` sub-function: `y`, `x` and `ry`. `y` is an incomplete data vector of length n, the dependent variable in the imputation regression model and `x` is an $n \times p$ matrix of predictors, those variables that were specified via the respective row in the `predictorMatrix`. `ry` is the response indicator of vector `y`, indicating if a value in `y` has been observed (`ry` = `TRUE`), or not (`ry` = `FALSE`).

2.3 Count data modeling in a nutshell

Before we introduce the functions of the **countimp** package, we familiarize the reader with typical count data models. A good introduction and summary may be found in Zeileis et al. (2008): The standard procedure to analyze ordinary count data is to fit a Poisson model under the generalized linear modeling (GLM) framework (Nelder & Wedderburn, 1972). GLMs describe the dependence

Table 2: Specification of imputation models in **mice**: The `predictorMatrix` argument.

	V1	V2	V3	V4	V5	V6	V7
V1	0	1	1	1	1	1	1
V2	1	0	1	1	1	0	1
V3	0	0	0	0	0	0	0
V4	0	0	0	0	0	0	0
V5	0	0	0	0	0	0	0
V6	0	0	0	0	0	0	0
V7	0	0	0	0	0	0	0

Note. Each row i denotes the imputation model for incompletely observed variable V_i in the data set. The zeros and ones indicate, if variable V_j, with $j \in 1, \ldots k$, where k is the number of variables in the data set, is part of the imputation model of V_i (1 = yes, 0 = no).

of a scalar variable y_i on a set of regressors x_i. The conditional distribution of $y_i|x_i$ is a linear exponential family with probability density

$$f(y;\lambda,\delta) = exp\left(\frac{y\lambda - b(\lambda)}{\delta} + c(y,\delta)\right), \tag{1}$$

with λ being the canonical parameter that depends on x_i via a linear predictor, δ being a dispersion parameter, and $b(\cdot)$ and $c(\cdot)$ being functions that determine, which member of the family (e.g., Poisson) is used. The mean is determined by $\mathsf{E}[y_i|x_i] = \mu_i = b'(\lambda_i)$, the variance by $\mathsf{VAR}[y_i|x_i] = \phi b''(\lambda_i)$. The dependence of $\mathsf{E}[y_i|x_i] = \mu_i$ on x_i is specified via a link function $g(\cdot)$. The classical poisson model

$$f(y;\mu) = \frac{exp(-\mu)\mu^y}{y!} \tag{2}$$

with link function $g(\mu) = log(\mu)$ assumes that the variance $\mathsf{VAR}(\mu)$ is equal to the mean μ (thus dispersion parameter δ takes on the value 1).

A problem that often arises in real life is that the restriction of equidispersion of the classical Poisson model is violated. Very often empirical data are overdispersed, which means that the variance is larger in comparison to the mean. Analyzing overdispersed data using classical Poisson regression leads to an underestimation of the variation in the data and an overestimation of statistical significance (cf. Zeileis et al., 2008). The standard procedure for overdispersion is to fit either a Quasi-Poisson model or a Negative Binomial model:

Quasi-Poisson regression is identical to Poisson regression except for the fact that it relaxes the assumption of equidispersion of the Poisson model. Here, dispersion parameter δ is estimated from the data rather than being fixed to 1. A second solution that leads to virtually identical results is to fit a Negative Binomial (NB) model. There is a number of different NB models and a number of different ways to parametrize a Negative Binomial model (cf Hilbe, 2007). What they have in common is that they all estimate a shape parameter that gives information about (over-)dispersion. Here it is labeled α:

$$f(y;\mu,\alpha) = \frac{\Gamma(y+\alpha)}{\Gamma(\alpha)y!}p^{\alpha}(1-p)^{y}, \tag{3}$$

with $p = \frac{\alpha}{\alpha+\mu}$. Typically the log is used as link function and $\mathsf{VAR}(\mu) = \mu + \frac{\mu^2}{\alpha}$. Parameter δ is fixed to one, as α already addresses overdispersion (Venables & Ripley, 2002).

An excess of zero counts can be addressed by fitting a zero-inflated Poisson or Negative Binomial model (Lambert, 1992) or by fitting a hurdle model (Mullahy, 1986). These models contain a second model component that addresses the zero counts: Hurdle models combine a left truncated count component and a right-censored hurdle component.

$$f_h(y;x,z,\beta,\gamma) = \begin{cases} f_z(0;z,\gamma) & \text{if } y=0 \\ (1-f_z(0;z,\gamma))\cdot f_c(y;x,\beta)/(1-f_c(0;x;\beta)) & \text{if } y>0. \end{cases} \tag{4}$$

The corresponding mean regression relationship is given by

$$log(\mu_i) = x_i^T\beta + log(1-f_{zero}(0;z_i,\gamma)) - log(1-f_{count}(0;x_i;\beta)). \tag{5}$$

Parameters β, γ and α are estimated by maximum likelihood and the count and hurdle components can be maximized separately, which also means that a different set of predictors could be used in each of the two model components. The count model family can be Poisson or NB with a log link, the zero hurdle model family is typical binomial with either a logit or probit link.

Zero-inflation models combine a point mass at zero $I_{\{0\}}(y)$ and the count component $f_{count}(y;x,\beta)$. Unlike hurdle models they allow two possible sources of zeros. Zeros may arise both in the count part and the zero part of the model. The probability of a zero count is inflated with probability $\pi = f_{zero}(0;z,\gamma)$, the unobserved probability π of belonging to point mass component is modeled by a binomial GLM $\pi = g^{-1}(z^T\gamma)$. The zero-inflated density is

$$f_{zeroinfl}(y;x,z,\beta,\gamma) = \pi\cdot I_{\{0\}}(y) + (1-\pi)\cdot f_{count}(y;x,\beta) \tag{6}$$

Parameters β, γ and α are estimated by ML and different sets of predictors could be used for the count part and the zero part. Again, the count model can be either fit as a Poisson or NB model.

One typical way to analyze clustered or panel count data is to fit a generalized linear mixed effects model using the Poisson or Quasi-Poisson family (Schall, 1991). Mixed effects models estimate typical regression parameters like any other regression model, but additionally estimate variation and covariation of intercepts and slopes across clusters, groups or individuals.

2.4 Introducing between-imputation variability

The functions in the **countimp** package use the regression models described in the previous section to multiply impute missing data. To introduce between-imputation variability, we either follow the Bayesian regression approach, as it is described in Rubin (1987), or we fit the specified models to different bootstrap samples for each of the *m* imputations. Note that multiple imputation inference using Rubin's rules (Rubin, 1987) incorporates two variance components: the with-imputation component and the between-imputation component. The combination of the two sources of variation is supposed to reflect the additional uncertainty in parameter estimation due to missing data in an adequate way (see for example Schafer, 1997).

Bayesian regression The Bayesian regression principle stems from Rubin (1987), where he proposes a bayesian logistic regression procedure: Here, he fits a logit model to the data and computes $\hat{\theta}$ and $\widehat{\mathsf{VAR}}(\hat{\theta})$, the posterior mean and the posterior variance of θ. $\hat{\theta}$ is defined by

$$\Pi_{i\in obs} f(Y_i|X_i,\hat{\theta}) \geq \Pi_{i\in obs} f(Y_i|X_i,\theta) \quad \forall\, \theta, \tag{7}$$

$\widehat{\mathsf{VAR}}(\hat{\theta})$ is defined by the negative inverse of the second derivative matrix of the log-posterior distribution at $\theta = \hat{\theta}$

$$\widehat{\mathsf{VAR}}(\hat{\theta}) = -\left[\frac{\partial^2}{\partial\theta\partial\theta} log \Pi_{i\in obs} f(Y_i|X_i,\theta)\bigg|_{\theta=\hat{\theta}}\right]^{-1}. \tag{8}$$

He then draws new parameters θ^* from $N(\hat{\theta},\widehat{\mathsf{VAR}}(\hat{\theta}))$[1]. In the third step, for each case with missing data $i \in mis$ he calculates fitted values $p_i = logit^{-1}(X_i\theta^*)$. These predicted values are not returned directly, but an additional random component is used: He draws independent uniform (0,1) random numbers u_i, with $i \in mis$. If u_i is larger than the predicted value p_i, he imputes $Y_i = 0$, else $Y_i = 1$. These steps are repeated *m* times to obtain the *m* imputations.

The bootstrap variant Having to assume that parameters θ^* can be simulated from a normal distribution $N(\hat{\theta},\widehat{\mathsf{VAR}}(\hat{\theta}))$ can sometimes be implausible. A good alternative in these cases is to draw a new bootstrap sample Y^* from the original data set Y *m* times and fit the respective regression

[1] It must be noted that the assumption that parameters θ^* can be simulated from a normal distribution $N(\hat{\theta},\widehat{\mathsf{VAR}}(\hat{\theta}))$ is being discussed quite critically (cf. Rubin, 1987). If the user does not want to make that assumption, we recommend to use the bootstrap variant.

model to Y^* to obtain parameters θ^* respectively.

3 Multiple imputation of incomplete count data

We now describe the functions in the **countimp** package in detail and give some practical examples.

3.1 Multiple imputation of "ordinary" count data

The functions `mice.impute.pois(y, ry, x)` and `mice.impute.pois.boot(y, ry, x)` impute univariate missing data using either Bayesian Poisson regression or a bootstrap Poisson regression model respectively. The arguments `y`, `ry`, and `x` need not be specified. They are automatically obtained from `mice()`. The numeric vector `y` is the incomplete count variable that shall be multiply imputed. The assumption is that `y` is Poisson distributed. `ry` is the response pattern of `y`, where `ry = TRUE` indicates an observed value and `ry = FALSE` a missing value. `x` is the design matrix with `length(y)` rows containing complete covariates – the imputation model, as it has been specified via the `predictorMatrix` argument of `mice()`. The variables in `x` are used to predict missing information in `y`.

The Bayesian method consists of the following steps:

1. Fit the Poisson model, and find bhat, the posterior mean, and V(bhat), the posterior variance of model parameters `b`.
2. Draw new parameters `b.star` from N(bhat,V(bhat)).
3. Compute predicted scores `p` using `exp(x[!ry, ] %*% b.star)`.
4. Draw imputations from `rpois(n = length(p), lambda = p)`.

The bootstrap method draws a bootstrap sample from `y[ry]` and `x[ry,]` and consists of the following steps:

1. Fit the Poisson model to the bootstrap sample and get model parameters `b`.
2. Compute predicted scores `p` using `exp(x[!ry, ] %*% b)`.
3. Draw imputations from `rpois(n = length(p), lambda = p)`.

Both functions rely on the standard `glm.fit` function from the **stats** package and return a numeric vector of length `sum(!ry)` with imputations.

3.1.1 Example

We now demonstrate how to use these functions to impute incomplete Poisson distributed count data: First, we create a toy data set containing the dependent count variable y and three continuous predictors, labeled x1 – x3. b0 – b3 are the corresponding regression coefficients, with b0 being the intercept coefficient.

```
R> set.seed( 1234 )
R> b0 <- 1
R> b1 <- .75
R> b2 <- -.25
R> b3 <- .5
R> N <- 5000
R> x1 <- rnorm(N)
R> x2 <- rnorm(N)
R> x3 <- rnorm(N)
R> lam <- exp( b0 + b1 * x1 + b2 * x2 + b3 * x3 )
R> y <- rpois( N, lam )
R> POIS <- data.frame( y, x1, x2, x3 )
```

We then introduce MAR missingness in y using the following function:

```
R> generate.md <- function( data, pos = 1, Z = 2, pmis = .5,
+   strength = c( .5, .5 ) )
+ {
+   total <- round( pmis * nrow(data) )
+   sm <- which( data[,Z] < mean( data[,Z] ) )
+   gr <- which( data[,Z] > mean( data[,Z] ) )
+   sel.sm <- sample( sm, round( strength[1] * total ) )
+   sel.gr <- sample( gr, round( strength[2] * total ) )
+   sel <- c( sel.sm, sel.gr )
+   data[sel,pos] <- NA
+   return(data)
+ }
R> MPOIS <- generate.md( POIS, pmis = .2, strength = c( .2, .8) )
```

The argument pos of generate.md() determines, which variable(s) in data shall receive missing values. Z is “the cause of missingness”, pmis is the percentage of missing data that shall be created, and `strength` determines, if a MCAR or MAR mechanism (e.g. Rubin, 1987) shall be created. `strength` takes two values between 0 and 1. They denote the percentages of incomplete cases that have values either above or below the mean of `Z`. `strength = c(.5, .5)` would thus simulate MCAR missingness – missingness not depending on `Z`, whereas `strength = c(.2, .8)` for example would generate MAR missingness: 20% of the cases with missing values in `pos` will be sampled from the cases who have a `Z` value smaller than `mean(Z)`, and 80% of the cases with missing values in `pos` will be sampled from the cases who have a `Z` value larger than `mean(Z)`.

Now, we have our example data set and can impute missing values in `y` using `mice()` together with `mice.impute.pois()`, do the repeated data analysis using the `with()` statement, and print a summary of the combined results using the functions `summary()` and `pool()`. Further details about using the `with()` and `pool()` functions may be found in van Buuren & Groothuis-Oudshoorn (2011).

```
R> require( "mice" )
R> imp <- mice( MPOIS, method = c( "pois" ,"" ,"" ,"" ), print = FALSE,
+  seed = 1234)
R> res <- with( imp, glm( y ~ x1 + x2 + x3, family = "poisson" ) )
R> print( pool.res <- summary( pool( res ) ) )
```

```
                   est          se         t        df Pr(>|t|)      lo 95
(Intercept)  0.9782221 0.010485964  93.28871 135.74447        0  0.9574851
x1           0.7539451 0.008195521  91.99477  83.23373        0  0.7376452
x2          -0.2366734 0.009052450 -26.14468  28.85129        0 -0.2551919
x3           0.4886071 0.008062132  60.60520  68.28584        0  0.4725206
                 hi 95 nmis       fmi    lambda
(Intercept)  0.9989591   NA 0.1808110 0.1688297
x1           0.7702450    0 0.2350376 0.2168748
x2          -0.2181549    0 0.4101527 0.3706337
x3           0.5046936    0 0.2611679 0.2398408
```

Alternatively, we can use the bootstrap variant, which yields similar results:

```
R> require( "mice" )
R> impBS <- mice( MPOIS, method = c( "pois.boot" ,"" ,"" ,"" ), print = FALSE,
+  seed = 1234 )
R> resBS <- with( impBS, glm( y ~ x1 + x2 + x3, family = "poisson" ) )
R> print( pool.resBS <- summary( pool( resBS ) ) )
```

```
                    est          se         t        df Pr(>|t|)      lo 95
(Intercept)  0.9820759 0.010298321  95.36272 190.77899        0  0.9617627
x1           0.7561635 0.007925848  95.40474 139.73054        0  0.7404934
x2          -0.2331646 0.009226357 -25.27158  25.20628        0 -0.2521588
x3           0.4878314 0.008131762  59.99086  60.03022        0  0.4715656
                  hi 95 nmis       fmi    lambda
(Intercept)  1.0023891   NA 0.1504005 0.1415403
x1           0.7718336    0 0.1780120 0.1663303
x2          -0.2141705    0 0.4394685 0.3966901
x3           0.5040971    0 0.2796465 0.2560400
```

3.2 Multiple imputation of overdispersed count data

The following functions may be used to impute incomplete overdispersed count data:

```
mice.impute.qpois(y, ry, x)
mice.impute.qpois.boot(y, ry, x)
mice.impute.nb(y, ry, x)
mice.impute.nb.boot(y, ry, x)
```

The functions impute univariate missing data based on either a Bayesian Quasi-Poisson (`"qpois"`) or Negative Binomial (`"nb"`) model, or by a boostrap Quasi-Poisson or Negative Binomial model (functions ending with "`.boot`"), and take the same arguments as the functions presented in Section 3.1. The imputation algorithms also work analogously, with the following differences: In the first step of the algorithm, the Quasi-Poisson imputation functions use `glm.fit()` with `family = "quasipoisson"` to fit the model, whereas the Negative Binomial imputation functions use `glm.nb()` from package **MASS**. In the last step, imputations are drawn from a NB distribution. The functions return a numeric vector of length `sum(!ry)` containing these imputations.

3.2.1 Example

We recycle some of the data from the example given in Section 3.1, namely `x1`–`x3` and `lam`, create an overdispersed count variable, and delete some of the data therein.

```
R> y.nb <- rnegbin( N, theta = 2, lam )
R> NB <- data.frame( y.nb, x1, x2, x3 )
R> MNB <- generate.md( NB, pmis = .2, strength = c( .2, .8) )
```

The function `generate.md()` has already been defined and explained in Section 3.1. Our toy data set containing the overdispersed count variable `y.nb` is now ready and we can impute missing data in `y.nb` using for example either the Quasi-Poisson or the NB approach. We use the `with()` statement to automate the repeated data analysis and combine the `m` results into an overall result using the `pool()` function.

```
R> require("mice")
R> imp1 <- mice( MNB, method = c( "qpois" ,"" ,"" ,"" ), print = FALSE,
+  seed = 1234 )
R> imp2 <- mice( MNB, method= c( "nb","" ,"" ,"" ), print = FALSE,
+  seed = 1234 )
R> res1 <- with( imp1, glm.nb( y.nb ~ x1 + x2 + x3 ) )
R> res2 <- with( imp2, glm.nb( y.nb ~ x1 + x2 + x3 ) )
```

Quasipoisson imputation yields the following results:

```
> print(pool.res1 <- summary( pool( res1 ) ) )

                    est          se         t        df Pr(>|t|)      lo 95
(Intercept)  1.0144907 0.01510831  67.14785 210.73328        0  0.9847080
x1           0.7677251 0.01481959  51.80476 472.83022        0  0.7386047
x2          -0.2060671 0.01570442 -13.12160  58.04839        0 -0.2375023
x3           0.4702718 0.01461203  32.18388 194.84819        0  0.4414537
                 hi 95 nmis       fmi     lambda
(Intercept)  1.0442735   NA 0.1424728 0.13437278
x1           0.7968455    0 0.0909138 0.08707662
x2          -0.1746318    0 0.2846613 0.26043236
x3           0.4990898    0 0.1486846 0.13999094
```

The combined results of NB imputation are highly similar:

```
> print(pool.res2 <- summary( pool( res2 ) ) )

                    est          se         t        df Pr(>|t|)      lo 95
(Intercept)  1.0141803 0.01513732  66.99868 357.96326        0  0.9844111
x1           0.7617044 0.01543719  49.34215 268.09241        0  0.7313108
x2          -0.2207702 0.01656256 -13.32947  43.89631        0 -0.2541521
x3           0.4721762 0.01479296  31.91899 270.21781        0  0.4430521
```

```
                hi 95 nmis       fmi    lambda
(Intercept)  1.0439496   NA 0.1063839 0.1014050
x1           0.7920979    0 0.1248754 0.1183711
x2          -0.1873883    0 0.3298207 0.2999661
x3           0.5013003    0 0.1243338 0.1178766
```

Unfortunately, the pool command in **mice** does not include the dispersion parameter. Therefore, we help ourselves using the following code:

```
R> EST1 <- EST2 <- SE1 <- SE2 <- vector( length = 5, "list" )
> for (i in 1:5) {
+   EST1[[i]] <- res1$analyses[[i]]$theta
+   SE1[[i]] <- res1$analyses[[i]]$SE.theta
+   EST2[[i]] <- res2$analyses[[i]]$theta
+   SE2[[i]] <- res2$analyses[[i]]$SE.theta
+ }
R> res.alpha <- rbind( miinference ( EST1, SE1 ), miinference( EST2, SE2 ) )
R> row.names( res.alpha ) <- c( "qpois", "nb" )
R> print( res.alpha )
```

```
      est      std.err    t.value  df       p.value lower    upper
qpois 2.030553 0.07907918 25.67747 218.0813 0       1.874696 2.186411
nb    1.877274 0.07927559 23.68035 37.77058 0       1.716757 2.037791
      r         fminf
qpois 0.1566468 0.1432531
nb    0.4824185 0.3585178
```

3.3 Multiple imputation of zero-inflated count data

The following functions have been designed to impute zero-inflated count data:

```
mice.impute.2l.zip(y, ry, x, type)
mice.impute.2l.zip.boot(y, ry, x, type)
mice.impute.2l.zinb(y, ry, x, type)
mice.impute.2l.zinb.boot(y, ry, x, type)
```

The functions impute univariate missing data using either a Bayesian regression ZIP (zero-inflated Poisson) or ZINB (zero-inflated Negative Binomial) or a bootstrap ZIP or ZINB regression model. The arguments `y`, `ry`, and `x` have already been explained in Section 3.1. `type` is a vector of length `ncol(x)` specifying the imputation model. It is automatically extracted from the `predictorMatrix` argument of `mice()` (see below for details). Note that despite the function name `.2l.` these functions do *not* fit a two-level model. The argument `.2l.` simply enables `mice`'s `type` argument that we need for the specification of more complex imputation models: Zero-inflation models are mixture models and require the specification of two models, a zero model, determining if the observational unit has a "certain zero" or not and a count model, determining, what count – both zero and non-zero – the observational unit has. The zero model is typically a logit model, the count model can be specified either as a Poisson or NB model. Note, that a different set of covariates (predictors) may be used for the zero and the count models. As already mentioned, the two models are specified via the `predictorMatrix` argument: allowed entries are '0', which means that the variable is not included in the imputation model and will thus be no part of `x`, '1' signifies that the variable will be included in both the zero *and* the count model. A '2' denotes a count model only variable, and finally '3' indicates a zero model only variable.

The Bayesian regression variants consist of the following steps:

1. Fit the ZIP or ZINB model, using the `zeroinfl()` function from package **pscl** and find bhat, the posterior mean, and V(bhat), the posterior variance of model parameters `b`.

2. Draw new parameters `b.star` from N(bhat,V(bhat)).

3. Compute predicted probabilities for observing each count `p`

4. Draw imputations from observed counts with selection probabilities `p`

The bootstrap functions (function names ending with "`.boot`") draw a bootstrap sample from `y[ry]` and `x[ry,]` and consist of the following steps:

1. Fit the ZIP or ZINB model to the bootstrap sample using the `zeroinfl()` function from package **pscl** and get model parameters `b`.

2. Compute predicted probabilities for observing each count `p`.

3. Draw imputations from observed counts with selection probabilities `p`.

All functions return a numeric vector of length `sum(!ry)` with imputations.

3.3.1 Example

Again, we present an example demonstrating the application of these functions. First, we simulate a data set with an incomplete zero-inflated count variable. We follow the procedure outlined in Erdman et al. (2008). In our example, coefficients `b0` – `b2` refer to the count model variables, `c0` and `c1` to the zero-model varibles, where subscript '0' stands for the intercept term. `theta` is the dispersion parameter.

```
R> b0 <- 1
R> b1 <- .3
R> b2 <- .3
R> c0 <- 0
R> c1 <- 2
R> theta <- 1
R> require("pscl")
R> set.seed(1234)
R> N <- 10000
R> x1 <- rnorm(N)
R> x2 <- rnorm(N)
R> x3 <- rnorm(N)
R> mu <- exp( b0 + b1 * x1 + b2 * x2 )
R> yzinb <- rnegbin( N, mu, theta)
R> pzero <- plogis( c1 * x3 )        # zero-infl. prob. depends on x3
R> ## Introduce zero-inflation
R> uni <- runif(N)
R> yzinb[uni < pzero] <- 0
R> zinbdata<-data.frame(yzinb,x1,x2,x3)
```

We then introduce MAR missingness. The function `generate.md` has been defined in Section 3.1.

```
R> zinbmdata <- generate.md( zinbdata, pmis = .3, strength = c( .2, .8) )
```

Having simulated our example data set, we are ready to specify the imputation model and impute the data. In the `predictorMatrix`, we set variables `x1` and `x2` as count model predictors and `x3` as a zero-model predictor.

```
R> ini <- mice( zinbmdata, m = 5, maxit = 0)
R> pred <- ini$predictorMatrix
```

```
R> pred[1,] <- c(0, 2, 2, 3)
R> meth<-ini$method
R> meth[1] <- "21.zinb"
R> imp.zinb <- mice( zinbmdata, m = 5, maxit = 1, method = meth,
+  predictorMatrix = pred, seed = 1234, print = FALSE)
```

Like in the previous examples, the `with` statement is then used to automate the repeated data analysis process. We fit the ZINB model using the `zeroinfl` function from package **pscl**, defining `x1` and `x2` as the count model variables by writing them before the "`|`", and `x3` as the predictor in the zero-model by writing `x3` after the "`|`" in the model formula. The `m` sets of regression parameters are finally combined into an overall result by the `pool` function from package **mice**:

```
R> res.zinb <- with( imp.zinb,
+  zeroinfl( yzinb ~ x1 + x2 | x3, dist = "negbin", link = "logit" ) )
R> summary( pool( res.zinb ) )

                          est         se         t         df     Pr(>|t|)
count_(Intercept)  0.97491260 0.02798674 34.834800   21.80435 0.000000e+00
count_x1           0.30530880 0.02377065 12.843939   18.47107 1.190474e-10
count_x2           0.30223258 0.01776991 17.008106  823.93486 0.000000e+00
zero_(Intercept)  -0.07953993 0.05735227 -1.386866 1342.16767 1.657128e-01
zero_x3            2.20015035 0.08879445 24.778017  113.35246 0.000000e+00
                       lo 95      hi 95 nmis        fmi     lambda
count_(Intercept)  0.9168414 1.03298376   NA 0.47365493 0.42749312
count_x1           0.2554596 0.35515794   NA 0.51442698 0.46455061
count_x2           0.2673530 0.33711221   NA 0.06878512 0.06652745
zero_(Intercept)  -0.1920498 0.03296991   NA 0.05199425 0.05058266
zero_x3            2.0242385 2.37606225   NA 0.20051971 0.18653697
```

All that remains to be done is to get a combined estimate of the dispersion parameter. We obtain that using the following code:

```
R> EST <- SE <- vector( length = imp.zinb$m, "list" )
R> for ( i in 1:imp.zinb$m ){
+   EST[[i]] <- log( res.zinb$analyses[[i]]$theta )
+   SE[[i]] <- res.zinb$analyses[[i]]$SE.logtheta
+ }
> print( data.frame( miinference( EST, SE ) ) )
```

```
         est    std.err   t.value       df   p.value       lower     upper
1 0.04369223 0.05847026 0.7472556 42.56254 0.4590178 -0.07425935 0.1616438
          r     fminf
1 0.4420869 0.3369995
```

Note that `zeroinfl()` gives an estimate of `theta`, but an estimate of `SE.logtheta`, the corresponding standard error on the log scale. We thus also need to use `log(theta)` for pooling.

3.4 Multiple imputation of incomplete "ordinary" two-level count data

The following functions can be used to multiply impute incomplete two-level Poisson data:

```
mice.impute.2l.poisson(y, ry, x, type, intercept = TRUE)
mice.impute.2l.poisson.noint(y, ry, x, type, intercept = FALSE)
mice.impute.2l.poisson.boot(y, ry, x, type, intercept = TRUE)
mice.impute.2l.poisson.noint.boot(y, ry, x, type, intercept = FALSE)
```

The functions impute missing data on the basis of a linear mixed effects Poisson model and are available as Bayesian regression variants or bootstrap regression variants (function ending with"`.boot`"). Functions with the term "`.noint`" do *not* include the intercept term as a random effect. The other functions automatically include the intercept term among the random effects.

The functions use the following arguments:

- `y`, a numeric vector with incomplete data in long format, which means that the data of the different groups or the repeated measurements are stacked upon each other.
- `ry`, the response pattern of `y`, with `TRUE` indicating that the respective value in `y` has been observed, and `FALSE` indicating a missing value.
- `x`, the design matrix with `length(y)` rows containing complete covariates, also in long format. `x` contains the variables in the imputation model, as it has been specified via the `predictorMatrix` argument of `mice()`.
- `type` is a vector of length `ncol(x)` specifying the imputation model and determining, which variables will be included in `x`. `type` is automatically extracted from the `predictorMatrix` argument of `mice()`. Allowed entries in the `predictorMatrix` are:
  - 0 = variable is not included in the imputation model (and will thus not be included in `x`).

  - 1 = variable will be included as a fixed effect.
  - 2 = variable will be included both as a fixed *and* random effect.
  - -2 = class variable. Please note that only one class variable is allowed.

- `intercept` determines, if the model will include the intercept as a random effect (`intercept = TRUE`) or not (`intercept = FALSE`).

`y`, `ry`, `x` and `type` are automatically passed on to the functions by `mice()`.

The imputation algorithm of the Bayesian regression variants works in the following way:

1. Fit the Poisson model using the `glmmPQL()` function from package **MASS**. This function fits a generalized linear mixed model with multivariate normal random effects, using penalized quasi-likelihood, as it is for example described in Schall (1991).
2. Find bhat, the posterior mean, and V(bhat), the posterior variance of model parameters `b`.
3. Draw `b.star` from N(bhat,V(bhat)).
4. Compute predicted counts `p`.
5. Draw imputations from Poisson distribution with mean `p`.

The bootstrap functions draw a bootstrap sample from `y[ry]` and `x[ry,]` and consist of the following steps:

1. Fit the glmmPQL model to the bootstrap sample and get model parameters `b`.
2. Compute predicted counts `p`.
3. Draw imputations from Poisson distribution with mean `p`.

All functions return a numeric vector of length `sum(!ry)` with imputations. We now demonstrate how to use these functions.

3.4.1 Example 1

We first load the example data set `NB.data`, which contains simulated two-level data.

```
R> data( "NB.data" )
R> head( NB.data )
```

```
   Y X0         X1 ID GRP Z0         Z1
1 NA  1 -0.5519079  1   1  1 -0.2380342
2  0  1  0.2026189  2   1  1 -0.2380342
3 NA  1  0.4324042  3   1  1 -0.2380342
4  2  1  0.1869166  4   1  1 -0.2380342
5  2  1 -0.9273682  5   1  1 -0.2380342
6 NA  1 -2.4500163  6   1  1 -0.2380342
```

`Y` is the incomplete count variable. `Y` is in fact overdispersed – but let us ignore that fact for a moment. In Section 3.5, we will impute and analyze these data with the "appropriate" functions for overdispersed two-level count data. `X1` is a continuous individual level predictor, `Z1` a continuous group level predictor. `ID` is the participant identifier and `GRP` indicates group membership. Missingness is MAR and depends on `Z1`. In fact, missingness has been introduced according to the following rule:

```
R> pmis <- plogis( -1 + NB.data$Z1 ) ## missingness probability
R> unif <- runif( nrow( NB.data ) ) ## random uniform numbers
R> sel <- which( unif < pmis )
R> NB.data$Y[sel] <- NA
```

This created around 34% missing data in `Y`.

We then set up the imputation:

```
R> ini <- mice( NB.data, maxit = 0 )
R> pred <- ini$predictorMatrix
R> pred[1,] <-c ( 0, 0, 2, 0, -2, 0, 1 )
R> meth <- ini$method
R> meth[1] <- "2l.poisson"
```

Running `mice()` with `maxit=0` sets up the `method` vector and `predictorMatrix`. We simple have to change the entries to meet our needs. In this example, we use `X1` as a fixed and random effect, and `Z1` as a fixed effect (group level predictor). The class variable is `GRP`. We then want to impute missing data in `Y` using the `mice.impute.2l.poisson()` function. The imputations are created by calling `mice()`:

```
R> imp <- mice( NB.data, maxit = 1, method = meth, pred = pred,
+  seed = 1234, print = FALSE )
```

We then do the automated repeated data analysis using the `do.mira()` function and obtain a summary of the results. We wrote the `do.mira()` function to automate repeated two-level linear mixed effects modeling. The function uses the following arguments: `imp` is an object of class `mids` with multiple imputations. `DV` is the dependent varible in the two-level regression model. `fixedeff` define the fixed effects in typical R formula language (see helpfile of function `glmmPQL()` for details). `randeff` defines the random effects in typical R formula language, `grp` is the class variable. The function automatically builds the required model formula for the random effects in the form `paste( "~", randeff, "|", grp, sep = "" )`. `id` is the observational unit identifier, `fam` defines the GLM family. Currently, we support `fam = "poisson"` and `fam = "nbinom"` for the Poisson or Negative Binomial model respectively. `do.mira()` reads out a table of combined fixed effects estimates, the pooled random effects standard deviations (simply the mean of the `imp$m` estimates), and a table of the combined random effects correlation(s) (again, simply the mean of the `imp$m` estimates) .

```
R> result<-do.mira( imp = imp, DV = "Y",
+         fixedeff = "X1+Z1", randeff = "X1",
+         grp = "GRP", id = "ID", fam = "poisson")
R> summary(result)

 Pooled Fixed Effects Coefficients:
                est std.err t.value    df p.value
(Intercept)  0.9957  0.0752 13.2354 191.7 < 2e-16 ***
X1           0.7841  0.0384 20.4041 535.2 < 2e-16 ***
Z1           0.4508  0.0920  4.8988  85.4 4.5e-06 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
                lower     upper          r      fminf
(Intercept) 0.8473191 1.1440901 0.16886269 0.15325777
X1          0.7085758 0.8595471 0.09463658 0.08984987
Z1          0.2678473 0.6337571 0.27621702 0.23416390

 Pooled Random Effects SD(s):
(Intercept)          X1    Residual
  0.4669164   0.2530191   1.5646265

 Pooled Random Effects Correlation(s):
            (Intercept)      X1
```

```
(Intercept)      1.0000 -0.3642
X1              -0.3642  1.0000
```

3.4.2 Example 2

Now, let us assume, we only want to use a random intercept model for data imputation and data analysis. The code changes thus:

```
R> ini <- mice( NB.data, maxit = 0 )
R> pred2 <- ini$predictorMatrix
R> pred2[1,] <-c ( 0, 0, 1, 0, -2, 0, 1 )
R> meth <- ini$method
R> meth[1] <- "2l.poisson"
R> imp2 <- mice( NB.data, maxit = 1, method = meth,
+  pred = pred2, seed = 1234, print = FALSE)
R> result2 <- do.mira( imp = imp, DV = "Y",
+  fixedeff = "X1+Z1", randeff = "1",
+  grp = "GRP", id = "ID", fam = "poisson")
R> summary(result2)
```

```
Pooled Fixed Effects Coefficients:
               est std.err t.value     df p.value
(Intercept)  1.0165  0.0726 13.9978 149.36 < 2e-16 ***
X1           0.8019  0.0151 53.2777   6.36 1.1e-09 ***
Z1           0.4548  0.0934  4.8687  89.04 4.8e-06 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
               lower     upper         r     fminf
(Intercept) 0.8730346 1.1600276 0.1956686 0.1746263
X1          0.7655931 0.8382532 3.8312561 0.8372394
Z1          0.2692076 0.6404418 0.2689618 0.2290785

 Pooled Random Effects SD(s):
(Intercept)    Residual
  0.4422699   1.7309927

 Pooled Random Effects Correlation(s):
[1] 0
```

Note that an intercept only model is estimated by specifying `randeff = "1"` when calling `do.mira()`.

3.4.3 Example 3

A random slope model without a random intercept could be fit thus:

```
R> ini <- mice( NB.data, maxit = 0 )
R> pred3 <- ini$predictorMatrix
R> pred3[1,] <-c ( 0, 0, 2, 0, -2, 0, 1 )
R> meth3 <- ini$method
R> meth3[1] <- "2l.poisson.noint"
R> imp3 <- mice( NB.data, maxit = 1, method = meth3,
+  pred = pred3, seed = 1234)
R> result3 <- do.mira( imp = imp, DV = "Y",
+  fixedeff = "X1+Z1", randeff = "0+X1",
+  grp = "GRP", id = "ID", fam = "poisson")
R> summary(result3)
```

```
Pooled Fixed Effects Coefficients:
                est std.err t.value     df p.value
(Intercept)  1.1117  0.0333 33.3614   5.04 4.1e-07 ***
X1           0.7354  0.0404 18.1834 593.74 < 2e-16 ***
Z1           0.4453  0.0421 10.5788   4.63   2e-04 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
                lower     upper         r      fminf
(Intercept) 1.0262949 1.1971789  8.139917 0.91779551
X1          0.6559541 0.8148096  0.089418 0.08515512
Z1          0.3344417 0.5560868 13.111031 0.94770084

 Pooled Random Effects SD(s):
       X1  Residual
0.2670139 1.8223524

 Pooled Random Effects Correlation(s):
[1] 0
```

Note that a random slope model without the intercept is estimated by specifying `randeff = "0+..."` when calling `do.mira()`.

3.5 Multiple imputation of incomplete overdispersed two-level count data

The following functions were designed to impute incomplete overdispersed two-level count data:

```
mice.impute.2l.nb2(y, ry, x, type, intercept = TRUE)
mice.impute.2l.nb2.boot(y, ry, x, type, intercept = TRUE)
mice.impute.2l.nb2.noint(y, ry, x, type, intercept = FALSE)
mice.impute.2l.nb2B.noint.boot(y, ry, x, type, intercept = FALSE)
```

These functions multiply impute an incomplete count variable in long format (i.e. the groups or measurement timepoints are stacked upon each other) based on a two-levle generalized linear mixed effects model – a NB2 model in the terminology of Hilbe (2007). The functions either follow the Bayesian Regression approach or the bootstrap approach (function name ending with "`.boot`"). The "`.noint`" variants treat the intercept only as a fixed, but *not* as a random effect.

The functions use the same arguments as the functions presented in Section 3.4. The algorithm also works quite similarly: To get an estimate of α, the dispersion parameter of the NB2 model, we use the `glmmadmb()` function of package **glmmADMB**. Note that **glmmADMB** is *not* available from `http://cran.r-project.org`. The package can be obtained from `http://glmmadmb.r-forge.r-project.org`. Information regarding the installation of that package can also be obtained from that website. Unfortunately, it is *not* possible to get estimates of random effects covariances with the current version of `glmmadmb()` (0.7.4). We deem this a major shortcoming. Until this issue is addressed, we use `glmmadmb()` only to estimate dispersion. We then fit the actual NB2 model using the `glmmPQL()` function from package **MASS** with family `negative.binomial(alpha)`. Note that **MASS** at the moment has no own capabilities to get an estimate for α. The `negative.binomial` family in **MASS** requires an estimate of α as input – thus the workaround.

The Bayesian regression algorithms thus consist of the following steps:

1. Fit the NB2 model, first using `glmmadmb()` to get an estimate for α, then using `glmmPQL()`, as described above.

2. Find bhat, the posterior mean, and V(bhat), the posterior variance of model parameters `b`.

3. Draw `b.star` from N(bhat,V(bhat)).

4. Compute predicted counts p.

5. Draw imputations from a Negative Binomial distribution with mean p and dispersion α.

The bootstrap variants draw a bootstrap sample from y[ry] and x[ry,] and consist of the following steps:

1. Fit the NB2 model to the bootstrap sample, first using glmmadmb() to get an estimate for α, then using glmmPQL(), as described above.

2. Compute predicted counts p.

3. Draw imputations a from Negative Binomial distribution with mean p and dispersion α.

All functions return a numeric vector of length sum(!ry) with imputations.

3.5.1 Example

We now give an example regarding how to use these functions: We first load the data set NB.data, which contains the incomplete overdispersed count variable Y (see Section 3.4 for details about that data set).

We then set up the imputation:

```
R> ini <- mice( NB.data, maxit = 0 )
R> pred <- ini$predictorMatrix
R> pred[1,] <-c ( 0, 0, 2, 0, -2, 0, 1 )
R> meth <- ini$method
R> meth[1] <- "2l.nb2.boot"
```

Running mice() with maxit=0 sets up the method vector and predictorMatrix. We simple have to change the entries to meet our needs. In this example, we use X1 as a fixed and random effect, and Z1 as a fixed effect (group level predictor). The class variable is GRP. We then want to impute missing data in Y using the mice.impute.2l.nb2.boot() function – the bootstrap variant. The imputations are created by calling mice():

```
R> imp <- mice( NB.data, maxit = 1, method = meth, pred = pred,
   + print = FALSE, seed = 1234 )
```

We then do the automated repeated data analysis using the do.mira() function and obtain a summary of the results:

```
R> result<-do.mira( imp = imp, DV = "Y",
+        fixedeff = "X1+Z1", randeff = "X1",
+        grp = "GRP", id = "ID", fam = "nbinom")
R> summary(result)

 Pooled Fixed Effects Coefficients:
                est std.err t.value       df p.value
(Intercept)  1.0018  0.0694 14.4334 102630.0 < 2e-16 ***
X1           0.7771  0.0371 20.9533   4934.8 < 2e-16 ***
Z1           0.4523  0.0829  5.4578  30633.5 4.9e-08 ***
alpha        2.0634  0.0564 36.5589     26.4 < 2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
                lower     upper           r       fminf
(Intercept) 0.8657269 1.1377963 0.006282212 0.006262357
X1          0.7044190 0.8498397 0.029304835 0.028864019
Z1          0.2898540 0.6147084 0.011559075 0.011491526
alpha       1.9474598 2.1793002 0.636567250 0.430477372

 Pooled Random Effects SD(s):
(Intercept)          X1    Residual
  0.4645968   0.2490271   1.0000917

 Pooled Random Effects Correlation(s):
            (Intercept)      X1
(Intercept)      1.0000 -0.3262
X1              -0.3262  1.0000
```

3.6 Multiple imputation of incomplete zero-inflated and overdispersed two-level count data

The following functions were designed to impute incomplete zero-inflated (and overdispersed) two-level count data:

```
mice.impute.2l.zihnb(y,ry,x,type,intercept.c=TRUE,intercept.z=TRUE)
mice.impute.2l.zihnb.noint.zero(y,ry,x,type,intercept.c=TRUE,intercept.z=FALSE)
mice.impute.2l.zihnb.noint.count(y,ry,x,type,intercept.c=FALSE,intercept.z=TRUE)
mice.impute.2l.zihnb.noint.both(y,ry,x,type,intercept.c=FALSE,intercept.z=FALSE)
mice.impute.2l.zihnb.boot(y,ry,x,type,intercept.c=TRUE,intercept.z=TRUE)
mice.impute.2l.zihnb.noint.zero.boot(y,ry,x,type,intercept.c=TRUE,intercept.z=FALSE)
mice.impute.2l.zihnb.noint.count.boot(y,ry,x,type,intercept.c=FALSE,intercept.z=TRUE)
mice.impute.2l.zihnb.noint.both.boot(y,ry,x,type,intercept.c=FALSE,intercept.z=FALSE)
```

The functions impute zero-inflated (and overdispersed) multilevel count data based on a two-level NB hurdle model, either using a Bayesian Regression ("`.zihnb`") or a bootstrap approach ("`.boot`"). The "`.noint`" variants treat the intercept only as a fixed, but not as a random effect. It may be specified, if the intercept is excluded from the random effects only in the zero model ("`.noint.zero`"), the count model ("`.noint.count`"), or both models ("`.noint.both`"). Note again that hurdle models are mixture models and consist of two models: the zero model (here a binomial GLMM), determining, if the observational unit has a zero or non-zero value, and the count model (here a zero-truncated two-level NB model) determining, what kind of non-zero value the observational unit has.

The following arguments are used by these functions:

`y` A numeric vector with incomplete data in long format (i.e. the groups are stacked upon each other).

`ry` the response indicator of `y`.

`x` a matrix with `length(y)` rows containing complete covariates (also in long format).

`type` a vector of `length(ncol(x))` determining the imputation model.

`intercept.c` `TRUE`: model will include intercept as a random effect in the count model; `FALSE`: model will not use intercept as a random effect.

`intercept.z` `TRUE`: model will include intercept as a random effect in the zero model; `FALSE`: model will not use intercept as a random effect.

`y,ry,x` and `type` are obtained from `mice()`. `type` is extracted from the respective row of the `predictorMatrix`: Allowed entries are:

- -2 = class variable (only one class variable is allowed!).
- 0 = variable not included in imputation model.
- 1 = variable will be included as a fixed effect (zero *and* count model).

- 2 = variable will be included as a fixed *and* random effect (zero *and* count model).
- 3 = variable will be included as a fixed effect (count model only).
- 4 = variable will be included as a fixed *and* random effect (count model only).
- 5 = variable will be included as a fixed effect (zero model only).
- 6 = variable will be included as a fixed *and* random effect (zero model only).

The Bayesian regression variants (see Rubin, 1987, pp. 169–170) consist of the following steps:

1. Fit the zero model (a two-level binomial generalized linear mixed effects model), using the `glmmPQL` function from package **MASS** and find bhat, the posterior mean, and V(bhat), the posterior variance of model parameters b.
2. Draw `b.star` from N(bhat,V(bhat)).
3. Compute predicted probabilities for having a zero vs. non-zero count, using `b.star`.
4. Draw imputations from a Binomial distribution and "remember" cases, who are supposed to get a non-zero count later on.
5. Fit the count model (a zero-truncated 2L NB model) using the `glmmadmb` function from package **glmmADMB** and the `truncnbinom` family; find bhat, the posterior mean, and V(bhat), the posterior variance of model parameters b.
6. Draw `b.star` from N(bhat,V(bhat)).
7. Compute predicted counts and draw non-zero imputations (from step 4) from a zero-truncated NB distribution.

The bootstrap functions draw a bootstrap sample from `y[ry]` and `x[ry,]` and consist of the following steps:

1. Fit the zero model to the bootstrap sample.
2. Compute predicted probabilities for having a zero vs. non-zero count.
3. Draw imputations from a Binomial distribution and "remember" cases, who are supposed to get a non-zero count.
4. Fit the count model to the boostrap sample.

5. Compute predicted counts and draw non-zero imputations (from step 3) from a zero-truncated NB distribution.

All functions return a numeric vector of length `sum(!ry)` with imputations.

3.6.1 Example

Again, we give an example, which illustrates the use of these functions:

```
R> data( "MZINB.data.Rdata" )
R> ini <- mice( MZINB.data, maxit = 0 )
R> pred <- ini$predictorMatrix
R> pred[1,] <- c( 0, 0, 2, 0, -2, 0, 1 )
R> meth <- ini$method
R> meth[1] <- "2l.zihnb"
R> imp <- mice( MZINB.data, maxit = 1, method = meth,
   + predictorMatrix = pred, seed = 1234)
R> result <- do.mira( imp, DV = "Y", fixedeff = "X1+Z1", randeff = "X1",
   + fam = "truncnbinom", grp = "GRP", id = "ID" )
R> summary( result )
```

```
Pooled Fixed Effects Coefficients:
                      est std.err t.value     df p.value
(Intercept).zero   0.0649  0.0755  0.8602   28.9    0.40
X1.zero            0.4429  0.0399 11.1048 2754.8 < 2e-16 ***
Z1.zero            0.1049  0.0733  1.4319   19.5    0.17
(Intercept).count  0.8663  0.0780 11.1049  429.2 < 2e-16 ***
X1.count           0.7959  0.0390 20.3958   77.7 < 2e-16 ***
Z1.count           0.4456  0.0717  6.2133  139.2 5.6e-09 ***
alpha.count        0.9083  0.0615 14.7678   36.5 < 2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
                        lower     upper          r      fminf
(Intercept).zero  -0.08945051 0.2192930 0.59223598 0.41131303
X1.zero            0.36465704 0.5210512 0.03961445 0.03880251
Z1.zero           -0.04817742 0.2580075 0.82803987 0.50160149
(Intercept).count  0.71297064 1.0196306 0.10685837 0.10072310
```

```
X1.count          0.71816528 0.8735436 0.29354952 0.24609907
Z1.count          0.30378615 0.5873607 0.20415673 0.18122715
alpha.count       0.78362897 1.0329870 0.49473412 0.36484799

 Pooled Random Effects SD(s):
 (Intercept).zero          X1.zero     Residual.zero (Intercept).count
        0.3897336        0.2342796         0.9952241         0.4812296
         X1.count
        0.1961808

 Pooled Random Effects Correlation(s):
                (Intercept).zero X1.zero
(Intercept).zero           1.000   0.027
X1.zero                    0.027   1.000
```

Please note again, that the current version of `glmmadmb()` cannot yet estimate correlations between random effects (see Section 3.5). This issue is currently being adressed by the **glmmADMB** developers and will hopefully be available in the near future. We will update the **countimp** package accordingly. Until then, `do.mira()` reads out pooled random effects correlations only for the zero model.

4 Discussion

We present multiple imputation procedures for various types of incompletely observed count data that work as an add-on for the Multiple Imputation by Chained Equations package **mice** in R (van Buuren & Groothuis-Oudshoorn, 2011). Currently, our algorithms cover ordinary, overdispersed, zero-inflated and multilevel count data. The advantage of integrating our algorithms into **mice** is that **mice** offers great imputation functionalities and diagnostic checks to ascertain convergence of the imputation Gibbs sampler and to asses plausibility of the created imputations. Our routines are thus embedded in a very popular, flexible and powerful imputation package. Practitioners already familiar with **mice** do not have to learn yet another statistical tool.

Our software is available free of charge from `http://www.uni-bielefeld.de/soz/kds/`. An overview of current software developments, evaluations of our imputation procedures, empirical examples, and applications may also be found there.

By and large, our algorithms are expected to yield good results, if statistical assumptions are more or less met, mainly that the MAR assumption holds and that the assumed statistical models fit the data reasonably well, and their assumptions are met. The greater the model misspecifications are, the more bias is to be expected. Secondly, effects of model misspecifications of course will get worse, the more missing data there are in the data set. If there is only a moderate percentage of missing data and model fit is good, then imputations should be plausible and valid.

Limitations and future research Currently, our multilevel procedures allow only two hierarchical levels. They furthermore assume homoscedasticity. An interesting avenue for future software development will be to allow for greater modeling flexibility, especially regarding autocorrelation in error terms or heteroscedasticity. At the moment, a limitation is that some functions rely on the **glmmADMB** package, which does not allow the estimation of random effects correlations. The current version has an argument `corStruct` that can be set to `full`, which means that a positive definite matrix with all elements is being estimated, however, *that feature is not yet working*! We hope that will be remedied with the next update. A further limitation is that `glmmadmb()` appears to be quite slow in comparison to other multilevel modeling tools in R. An avenue for future program development might be to speed things up a little.

References

Erdman, D., Jackson, L., & Sinko, A. (2008). Zero-inflated poisson and zero-inflated negative binomial models using the COUNTREG procedure. In SAS Institute Inc. (Ed.), *Proceedings of the SAS® Global Forum 2008 Conference.* Cary, NC: SAS Institute Inc. Available from `http://www2.sas.com/proceedings/forum2008/322-2008.pdf`

Hilbe, J. M. (2007). *Negative binomial regression.* Cambridge: Cambridge University Press.

Kleinke, K., Stemmler, M., Reinecke, J., & Lösel, F. (2011). Efficient ways to impute incomplete panel data. *Advances in Statistical Analysis*, *95*(4), 351–373.

Lambert, D. (1992). Zero-inflated poisson regression. With an application to defects in manufacturing. *Technometrics*, *34*, 1–14.

Landerman, L., Land, K., & Pieper, C. (1997). An empirical evaluation of the predictive mean matching method for imputing missing values. *Sociological methods & research*, *26*(1), 3–33.

Mullahy, J. (1986). Specification and testing of some modified count data models. *Journal of Econometrics*, *33*, 341–365.

Nelder, J. A., & Wedderburn, R. W. M. (1972). Generalized linear models. *Journal of the Royal Statistical Society A*, *135*, 370–384.

Raghunathan, T. E., Lepkowski, J. M., van Hoewyk, J., & Solenberger, P. (2001). A multivariate technique for multiply imputing missing values using a sequence of regression models. *Survey Methodology*, *27*(1), 85–96.

Rubin, D. B. (1976). Inference and missing data. *Biometrika*, *63*, 581–592.

Rubin, D. B. (1987). *Multiple imputation for nonresponse in surveys*. New York: Wiley.

Schafer, J. L. (1997). *Analysis of incomplete multivariate data*. London: Chapman & Hall.

Schafer, J. L., & Graham, J. W. (2002). Missing data: Our view of the state of the art. *Psychological Methods*, *7*, 147–177.

Schafer, J. L., & Yucel, R. M. (2002). Computational strategies for multivariate linear mixed-effects models with missing values. *Journal of Computational and Graphical Statistics*, *11*(2), 437–457.

Schall, R. (1991). Estimation in generalized linear models with random effects. *Biometrika*, *78*(4), 719–727.

Su, Y.-S., Gelman, A., Hill, J., & Yajima, M. (2009). Multiple imputation with diagnostics (mi) in R: Opening windows into the black box. *Journal of Statistical Software*, *20*(1), 1–27.

van Buuren, S., Brand, J. P. L., Groothuis-Oudshoorn, C. G. M., & Rubin, D. B. (2006). Fully conditional specification in multivariate imputation. *Journal of Statistical Computation and Simulation*, *76*(12), 1049–1064.

van Buuren, S., & Groothuis-Oudshoorn, K. (2011). MICE: Multivariate imputation by chained equations in R. *Journal of Statistical Software*, *45*(3), 1–67.

Venables, W. N., & Ripley, B. D. (2002). *Modern applied statistics with S*. New York: Springer.

Zeileis, A., Kleiber, C., & Jackman, S. (2008). Regression models for count data in R. *Journal of Statistical Software*, *27*(8), 1–25.