

countimp version 3

Kristian Kleinke and Jost Reinecke

2026-09-27

Abstract

Count data are non-negative integer values and give the frequency of occurrence of a certain event or behaviour within a given timespan. Count data are usually not normally distributed but are often skewed and require special analysis and imputation techniques. Yet, the imputation packages currently available provide only basic support: Poisson imputation is common and negative binomial imputation less so, while excess zeros, a hurdle at zero, truncation, censoring, a known upper bound and underdispersion are rarely offered as a model one can select. Furthermore, clustered counts can only be imputed under a Poisson model. The `countimp` package fills this gap and provides easy to use multiple imputation (MI) procedures for various kinds of incomplete count data based on either a Bayesian regression approach or on a bootstrap regression approach within a fully conditional specification (FCS) framework. Version 3 covers imputation of ordinary counts under the Poisson model, overdispersed counts under the negative binomial (NB) model, zero-inflated and hurdle variants of both, zero-truncated counts, right-censored counts, counts confined to a known range, and Conway-Maxwell-Poisson, which also covers underdispersion. Two-level and three-level data are generally supported; A few models that restrict the range of the count variable, i.e., truncated, censored, and bounded are single-level so far. Extending these imputation functions to support multilevel models is an avenue for future program versions. Imputation models may be specified either in the classical `mice` style through arguments `method` and `predictorMatrix`, or via an `lme4`-style formula in which a grouping term selects the multilevel method. This manual shows how to use the package; for the statistical background, the derivations and the simulation evidence, see Kleinke & Reinecke (2026).

Table of contents

1	About this manual	2
2	Statistical background	3
2.1	Multiple imputation frameworks	3
2.2	Bayesian regression and bootstrap regression	3
2.3	Count data models	4
3	Overview	5
3.1	What is new in version 3?	5
3.1.1	New model families	6
3.1.2	A formula interface	6
3.1.3	Variables on the cluster level	6
4	Choosing an imputation model for count data	6
4.1	What is recommended in the literature	6
5	countimp	7
5.1	A cautionary note	7
5.2	Installation	8
5.3	The example data	8
5.4	How <code>countimp</code> is organised	8
5.5	Two ways to specify the model	10
5.6	The imputation methods at a glance	11
5.7	Imputation of incomplete Poisson distributed variables	12
5.7.1	Example	13
5.8	Imputation of overdispersed count data	14

5.8.1	Imputation based on a quasi-Poisson model	15
5.8.2	Example	15
5.8.3	Imputation based on a negative binomial model	16
5.9	Imputation of count data with an excess number of zero counts	16
5.9.1	Example	17
5.9.2	Simulation evidence	18
5.10	Imputation of hurdle distributed count data	18
5.10.1	Simulation evidence	19
5.11	Imputation of zero-truncated count data	19
5.11.1	Simulation evidence	20
5.12	Imputation of underdispersed count data	20
5.12.1	Simulation evidence	21
5.13	Imputation of right-censored counts	21
5.13.1	Simulation evidence	23
5.14	Imputation of counts with a known upper bound	23
5.14.1	Simulation evidence	24
5.15	Imputation of two-level Poisson distributed data	24
5.15.1	Example	24
5.15.2	Three levels	25
5.15.3	Variables that live at the cluster level	26
5.16	Imputation of overdispersed two-level count data	27
5.16.1	Example	27
5.16.2	Simulation evidence	28
5.17	Imputation of zero-inflated two-level count data	28
5.17.1	Example	29
5.17.2	Simulation evidence	29
5.18	Choosing a model and checking the imputations	30
5.18.1	Choosing a family: <code>countimp_fit_diag()</code>	30
5.18.2	What happened during imputation: <code>countimp_diagnostics()</code>	30
5.18.3	Observed against imputed: <code>compare.obs.imp()</code> and <code>compare.percent.count()</code>	32
Contact		32
References		32

1 About this manual

Count data typically require special analysis and imputation techniques (for a comprehensive overview on models for various types of count data, see Hilbe, 2011).

Multiple imputation under a count model, however, is still unevenly supported by statistical packages. Poisson and, less often, negative binomial models are by now available in several programs (for an overview, see Section 3). What remains scarce is everything beyond them: an excess of zeros, a hurdle at zero, censoring, a known upper bound, underdispersion, and counts nested in clusters.

This document is a **user manual**: it shows which method to call for which kind of count variable, and what the call looks like. It is deliberately *not* another introduction to missing data and multiple imputation; book-length treatments of the theory are for example Kleinke et al. (2020), van Buuren (2018) and Little & Rubin (2002).

What the rest of the manual assumes of that theory is the basic principle of MI based on Rubin (1987): each missing value is filled in m times from a prediction model for the incomplete data given the observed data, the m completed data sets are analysed separately by any complete-data technique (here, typically a count model that fits the empirical data at hand), and the m sets of estimates and standard errors are combined by the rules in Rubin (1987) or Barnard & Rubin (1999), the variance *between* the imputed data sets reflecting the additional estimation uncertainty due to the missing data. Like most MI methods, the ones in this package assume the data are missing at random (MAR, Rubin, 1976), meaning missingness *can* be predicted from observed information and does not additionally depend on unobserved information.

This manual is structured as follows:

- Section 2 gives an overview on the models the package fits and lays out the terminology the rest of the manual uses. It is deliberately brief; the derivations, the distinction between zero-inflated and hurdle models, and the two ways of drawing parameters are treated in Kleinke & Reinecke (2026).
- Section 3 lists available software that imputes counts under some kind of count model, and says what version 3 of `countimp` adds on top of the already available packages.
- Section 4 sets out what the literature recommends for incomplete count data.
- Section 5 is the main part of this user’s manual: each model family has its own section, where we explain the syntax, give an example, and point to the simulation evidence for it. Detailed information about the simulation designs, the results and a discussion are given in Kleinke & Reinecke (2026), which is the citable source for the designs. An evaluation of the algorithms from Version 1 may furthermore be found in Kleinke & Reinecke (2013), Kleinke & Reinecke (2015a) and Kleinke & Reinecke (2015b); the results on proxy strategies from the Version 2 manual are published in Kleinke & Reinecke (2022).

In addition to this manual, a **vignette** comes with package `countimp`: `vignette("countimp")` covers choosing a model family, the formula interface, and how to check imputations. The manual covers both in Section 5.18, where `countimp_fit_diag()` weighs the candidate families against an observed variable before anything is imputed, `countimp_diagnostics()` reports what the methods did on the way to a draw, and `compare.obs.imp()` and `compare.percent.count()` put the observed and the imputed part side by side.

2 Statistical background

This section lays out the terminology the rest of the manual uses: the two frameworks for imputing multivariate incomplete data, the two ways of drawing the parameters of an imputation model, and the thirteen count model families `countimp` provides.

2.1 Multiple imputation frameworks

The two common frameworks are **joint modelling** (JM, e.g. Schafer, 1997) and **fully conditional specification** (FCS). The latter approach appears in the literature also as *conditional modelling* and as *multiple imputation by chained equations*, which is where the name `mice` comes from (van Buuren, 2018); this manual says FCS throughout. JM specifies one joint imputation model for the data set as a whole, whereas FCS specifies a separate conditional model for each incompletely observed variable. `countimp` follows the conditional approach, which was developed for exactly the situation this package addresses: a data set of variables of many different types, where an explicit joint model cannot easily be formulated (Raghunathan et al., 2001, p. 86).

2.2 Bayesian regression and bootstrap regression

Regression based imputation methods can be classified as either deterministic or stochastic, depending on whether or not there is some degree of randomness in the imputation process (van Buuren, 2018 calls the two *regression imputation* and *stochastic regression imputation*). A deterministic imputation approach would impute the value that is predicted for the respective incomplete case. Such approaches usually cause an underestimation of standard errors (Little & Rubin, 2002).

Multiple imputation based on Rubin’s theory (Rubin, 1987) includes two stochastic components that introduce an adequate amount of between-imputation variability to reflect the estimation uncertainty due to missing data. The first component is **parameter uncertainty**. The imputation model is itself estimated from finite data, so its parameters are not known quantities, and each of the m imputations has to use its own draw of them. This is what the Bayesian and the bootstrap variant below do in two different ways: **the predictive draw**. Secondly, an imputed count is still not the predicted mean μ^* but a value drawn from the count distribution around it. Omitting the second component (i.e., imputing μ^* itself) is the deterministic approach named above, and it is where the underestimated standard errors come from.

Two common solutions in FCS to address parameter uncertainty are Bayesian regression (see for example Rubin, 1987, pp. 169–170), and bootstrap regression: Bayesian regression first fits a model to the complete part of the data and computes the posterior mean $\hat{\beta}$ and posterior variance $V(\hat{\beta})$ of the model parameters β . Then, new parameters are simulated from $\mathcal{N}(\hat{\beta}, V(\hat{\beta}))$. These new parameters yield a predicted mean for every incomplete case, and the imputation is drawn from the count distribution with that mean.

The bootstrap variant draws a bootstrap sample from the observed cases, fits the regression model to the bootstrap sample and uses the obtained model parameters to predict the mean for the incomplete part of the data; the imputation is then drawn exactly as above. The two variants therefore differ in the *first* component only — a draw from the posterior of the parameters versus a refit on a resample of the observed cases — and not in the draw that produces the imputed count.

Imputation functions from package `countimp` come in a Bayesian regression and a bootstrap regression variant. The level-2-only count methods `2lonly.pois` and `2lonly.nb` are the exception: they currently draw from the posterior only.

Which of the two to use? It often doesn't make a difference. Across the 94 paired comparisons of a Bayesian against a bootstrap arm in the simulations behind this manual, only 3 differences in coverage exceed the Monte Carlo threshold for a difference between two rates. None favours the Bayesian arm. Furthermore, the bootstrap is the cheaper of the two: it refits the model on a resample, where the Bayesian draw needs the covariance matrix of the parameters, and it makes no distributional assumption about that matrix. One caveat, however, applies to clustered data — the cluster bootstrap resamples whole clusters, so it needs a substantial number of them, and little is known about the level-1 and level-2 sample sizes required. The two-level evidence here rests on 50 clusters; Kleinke & Reinecke (2026) varies them between 15 and 30 and finds the usual maximum-likelihood underestimation of variance components at the lower end, which affects both draws alike.

2.3 Count data models

`countimp` version 3 has thirteen model families. This section gives an overview on which family fits which scenario. The densities, the link functions and the distinction between zero-inflated and hurdle models are derived in Kleinke & Reinecke (2026); good references for the theoretical background are Zeileis et al. (2008) and, comprehensively, Hilbe (2011).

The variable is ...	Family	method name
equidispersed, variance $\approx$ mean	Poisson	<code>poisson</code>
overdispersed, variance $>$ mean	quasi-Poisson	<code>quasipoisson</code>
overdispersed	negative binomial (NB)	<code>nb</code>
underdispersed, variance $<$ mean	Conway-Maxwell-Poisson	<code>cmp</code>
zero-inflated	zero-inflated Poisson	<code>zip</code>
the same, and overdispersed	zero-inflated NB	<code>zinb</code>
zero or not zero with two separate processes	hurdle Poisson	<code>hp</code>
the same, and overdispersed	hurdle NB	<code>hnb</code>
never zero by construction	zero-truncated Poisson	<code>ztp</code>
the same, and overdispersed	zero-truncated NB	<code>ztnb</code>
recorded only up to a reporting limit	censored Poisson	<code>cp</code>
the same, and overdispersed	censored NB	<code>cnb</code>
confined to a known maximum	bounded Poisson / NB	<code>bp</code> / <code>bnb</code>

Overdispersion. The classical Poisson model assumes equidispersion, i.e., the variance equals the mean. Empirical count data very often have a larger variance. They are *overdispersed*. Fitting a Poisson model to them understates the standard errors. A quasi-Poisson model keeps the Poisson mean structure and multiplies the variance by a dispersion parameter. In contrast, the negative binomial model assumes a different distribution altogether (Poisson Gamma mixture).

Excess zeros: two different assumptions. A zero-inflated model [ZIP with a Poisson count part, ZINB with a negative binomial one; Lambert (1992)] assumes that zeros arise from *two* sources: a structural zero from the zero part, or a sampled zero from the count part. A hurdle model [HP and HNB respectively; Mullahy (1986)] in contrast assumes the zeros come *only* from the zero part, and models the positive counts as zero-truncated. Which one is right is a question about the theory and the data: is there for example a subgroup of respondents who could not have offended at all (zero-inflation), or does everyone face the same first-offence hurdle and then a count process models the number of offences (hurdle)? Kleinke & Reinecke (2026) treats this at length and shows by simulation what happens when the wrong one is chosen.

Our general advice is to always **prefer a model with fitting assumptions**. Based on Yu et al. (2007) and Kleinke (2017) we recommend imputation methods whose parametric assumptions fit the variable at hand, rather than using a proxy strategy. See Section 4 for what the two editions of van Buuren (2018) recommend, and what is documented about the alternatives.

3 Overview

This section gives an overview on software packages apart from `countimp` that offer count imputation models, and what version 3 of `countimp` adds on top of it.

Checked in September 2026, the following software supports count models:

- **IVEware**, available as a SAS add-on or as a standalone version, sorts every variable into one of five types (continuous, binary, categorical, count, semicontinuous) and picks the regression model from the type. Counts get an ordinary **Poisson** model (Raghunathan et al., 2001). No overdispersion, nor excess zeros are supported.
- **Stata** imputes counts with its own `mi impute: poisson` for a count variable and `nbreg` for an overdispersed one. Both are univariate methods and both are admitted inside `mi impute chained`, so they are available within fully conditional specification (FCS), the framework `mice` and `countimp` use as well (StataCorp, 2025). The user-written `ice`, named in the manual for version 2, predates these commands. Zero-inflation, hurdle and multilevel count models are not among the methods.
- R package `mi` imputes a variable declared as a count with a Bayesian **quasi-Poisson** GLM by default, fitted by `bayesglm` (Su et al., 2011); the ordinary Poisson is available and the package advises against it as overly restrictive. Zero-inflation, hurdle and multilevel count models are not supported.
- R package `micemd` (Audigier & Resche-Rigon, 2018) adds two **two-level Poisson** methods to `mice`: `2l.glm.pois`, a Bayesian mixed model with non-informative priors, and `2l.2stage.pois`, a two-stage estimator. Poisson only.
- R package `JointAI` (Erler et al., 2021) takes the joint-model route rather than FCS and estimates the analysis model and the imputation models together in a Bayesian framework. An incomplete count can be given a Poisson model, with or without random effects. Poisson only.
- R package `smcfcs` (Bartlett, 2025) imputes incomplete **covariates** compatibly with the substantive model and offers `poisson` among its methods. It imputes covariates; the outcome is the substantive model.
- **Blimp** (Keller & Enders, 2026) declares counts on a `COUNT:` line and fits a negative binomial regression; no further specification is needed. Two limits are worth knowing. An incomplete count variable can only be an **outcome, not a predictor**. And excess zeros are handled by writing out a **two-part model**, a probit model for a binary zero-versus-non-zero indicator beside the count model in the model section, not by selecting a zero-inflated family.
- **Mplus** (Muthén & Muthén, 2017) supports count imputation models since Version 8.5 (Muthén & Muthén, 2020). A count variable is marked (`ct`) on the `IMPUTE` list of `DATA IMPUTATION` and is imputed under the model written in the `MODEL` command. Two conditions come with it: `ESTIMATOR=BAYES`, and **H0 imputation only**, i.e., the imputation model has to be spelled out, the unrestricted H1 route does not accept counts. Under Bayes a count is Poisson or negative binomial (NB2 or Pólya-Gamma parameterisation), so the zero-inflated, hurdle and zero-truncated count models that `Mplus` fits by maximum likelihood are not available for imputation. Estimating an incomplete count model directly under FIML was possible before and remains so; what Version 8.5 added is the imputation.
- Two packages are very **domain-specific**: `accelmissing` (Lee, 2025) imputes accelerometer counts under a zero-inflated Poisson lognormal model, and `dejaVu` (Burkoff et al., 2024) imputes recurrent-event counts under a negative binomial model in a reference-based framework.

3.1 What is new in version 3?

`countimp` version 3 extends the functionality of version 2 and adds new model families, a second, more user-friendly way of specifying the imputation model (via an `lme4`-style formula), and methods for level-2 *and* level-3 variables.

3.1.1 New model families

- **Zero-truncated counts:** `ztp` and `ztnb` for variables that cannot be zero by construction, e.g. days in hospital among those admitted, or number of offences among convicted offenders.
- **Right-censored counts:** `cp` and `cnb` for counts recorded only up to a reporting limit (`sensor = 10`), e.g. a questionnaire whose top category reads “x or more”, like 10 or more criminal offences.
- **Counts confined to a known range:** `bp` and `bnb` for a variable that cannot exceed a known maximum (`bounds = c(0, 10)`), e.g. the number of correct answers in a ten-item test. Here ten means exactly ten, and eleven is impossible.
- **Conway-Maxwell-Poisson:** `cmp`, the only family that covers **underdispersion**; counts that vary *less* than a Poisson process would because they cluster tightly around a typical value, such as the check-ups a patient attends in a year when a fixed schedule is recommended. Where a hard maximum is what makes a count underdispersed, the bounded families above are the better choice (see Section 5.12)
- **Two-level Conway-Maxwell-Poisson:** `2l.cmp`, the same distribution for underdispersed counts that are clustered or repeated over time, specified in long format with a cluster indicator like the other two-level families.

Throughout, the endings `p` and `nb` denote the Poisson and the negative binomial variant of a family.

3.1.2 A formula interface

The imputation model may now be specified via an `lme4`-style formula and a family function:

```
countimp(data, formulas = list(y ~ x + z + (1 | id)), family = nb())
```

When a grouping term such as `(1 | id)` is included into the formula, `countimp` automatically selects the multilevel method, so users need not remember the exact name of the underlying imputation function. This is a huge advantage in comparison to version 2. The old route via arguments `method` and `predictorMatrix` is still supported for backward compatibility. Both ways of specifying the imputation model are shown side by side in Section 5.5.

3.1.3 Variables on the cluster level

Some variables do not vary within their cluster, e.g. a therapist’s years of experience is the same for all of his or her patients. In the long format they still occupy one cell per row, and an FCS engine would impute cell by cell. Thus, imputation of an incomplete level-two variable might result in different years of experience of *this* therapist for every patient. The `2lonly` functions like `2lonly.pois` and `2lonly.nb` impute such a variable at its level and ensure that the imputation is the same for all units within this cluster.

4 Choosing an imputation model for count data

Count data are non-negative integers, usually skewed, and often carry more zeros than a single count distribution accounts for. Previous missing data research discourages the use of imputation models that do not have a good fit to the distribution of the empirical data at hand (Kleinke, 2017; Yu et al., 2007).

For a long time such models were not available in general purpose imputation software, and the strategies practitioners fell back on were to ignore that the data are skewed non-negative integers and use standard procedures based on OLS regression, to apply some *normalising* transformation like a log or square root transformation followed by normal model multiple imputation and back-transformation, or to treat the data as *ordered* categorical. The transformation–imputation–backtransformation approach was implemented in Schafer’s `norm` for Windows software in 1999, when it was one of the very few user friendly solutions available and one of the few options practitioners had to handle missing data in non-normal variables. Today, missing data researchers (e.g. von Hippel, 2013) usually discourage the use of transformations to *make* the normality assumption of an imputation model more plausible, because better alternatives are available. Supplying those alternatives, and documenting how to use them, is what this package and this manual are for.

4.1 What is recommended in the literature

The first edition of one of the standard textbooks ‘Flexible Imputation of Missing Data’ (van Buuren, 2012, sec. 3.6.1) lists four strategies for count data:

- predictive mean matching (PMM)
- ordered categorical regression
- (zero-inflated) Poisson regression
- (zero-inflated) negative binomial regression,

and adds that it is not “yet clear whether one of the four methods consistently outperforms the others”; the advice given there is “to use predictive mean matching for count data” (p. 78).

The second edition (van Buuren, 2018, sec. 3.7.1) keeps the same four strategies but no longer issues that blanket recommendation. It points to Kleinke & Reinecke (2013) and to the `countimp` and `ImputeRobust` (Salfran & Spiess, 2018) packages as the parametric / semiparametric routes, and closes on Kleinke (2017) with the qualification that predictive mean matching can be recommended “for imputation of mildly to moderately skewed variables when sample size is sufficiently large”. Section 3.4.5 of the same edition names the two structural limits behind that qualification: predictive mean matching cannot extrapolate beyond the range of the observed data, and it cannot interpolate within that range where the interior is sparse.

Two of the four strategies on that list are count models, and they are the ones this package supplies. `countimp` implements Poisson and negative binomial regression and their zero-inflated counterparts, and — beyond the textbook’s list — hurdle, zero-truncated, bounded, right-censored and Conway-Maxwell-Poisson models, each with a single-level and a two-level version (Section 5.6). This manual does *not* take a position against van Buuren’s textbook. It documents the textbook’s third and fourth strategy, made available in a form that can be used from `mice` or from `countimp`’s own engine.

For the categorical strategy the evidence is less encouraging, and it is worth stating here because the strategy is still in use. Wu et al. (2015) examined the proportional odds model among others and advise against its application; van Buuren (2018), Section 3.6.3, reviews this and further evidence and recommends `pmm`, `cart` or `rf` instead whenever the number of parameters to be estimated is large relative to the number of cases per category. None of this work, however, addresses zero-inflated Poisson or negative binomial data.

Regarding predictive mean matching, our position is that it remains a good all-round method for count data that are not too severely skewed. See for example Kleinke (2017), where we simulated incomplete Poisson distributed data, found acceptable results when skewness was only mild to moderate and when not too many data had to be imputed; inferences were biased once the counts were more heavily skewed and the missing data percentage was substantial. The mechanism is the one listed under the pitfalls of the method (van Buuren, 2018, sec. 3.4.4): in the sparse upper tail of a skewed count distribution there are few donors to match, and the same small observed values are drawn over and over. Chapter 5 of Kleinke et al. (2020) works through the practical side of this on empirical panel data, including the diagnostic tools with which imputations of a count variable can be inspected.

What follows in Section 5 is a user’s manual for `countimp`: for each model family, we explain the syntax, give a worked example, and point to the simulation evidence for that family. Those evaluations hold a `countimp` method against the model that generated the data — they establish whether the method recovers what it is meant to recover. They are not comparisons against imputation methods from other packages. Which of the four strategies above to prefer in a given applied scenario is a research question of its own and beyond the scope of this manual.

5 `countimp`

5.1 A cautionary note

Package `countimp` creates imputations based on parametric count models. *Plausibility* of the obtained imputations will depend upon the correct specification of the imputation procedure, i.e. the count data regression model that is used to predict missing information must fit the empirical data well.

Research by Kleinke & Reinecke (2013) suggests that misspecifications of the (count) imputation model can bias statistical inferences quite noticeably. We therefore advise researchers to spend sufficient time on modelling. We would like to refer readers to the excellent book by Hilbe (2011) to determine which count model fits the data best.

We would also like to draw attention to R packages `Qtools` and `ImputeRobust`, which allow to create imputations based on nonparametric quantile regression and semi-parametric generalized additive models, which might be a suitable alternative in situations, where assumptions of parametric count models are violated.

5.2 Installation

The latest version can be installed from GitHub using function `install_github` from package `remotes`:

```
remotes::install_github('kkleinke/countimp', build_vignettes = TRUE)
```

5.3 The example data

Most examples in this chapter use `crim4w` and `crim4l`, which ship with the package. They are two shapes of the same data: four annual waves of self-reported delinquency from the CrimoC panel study of adolescents (Boers et al., 2010; Reinecke & Weins, 2013). Chapter 5 of Kleinke et al. (2020) works through the same data at length, with a substantive growth model and the missing-data inspection that precede any imputation.

`crim4w` is the data set in wide format: 2064 respondents, one row each. `ACRIM` to `DCRIM` are the delinquency counts of the four waves. `FEMALE` is the gender indicator, and `GY`, `RE`, and `HA` are dummies for the the top-level branch, the intermediate, and lowest-level branch of the German secondary school system. The covariates are complete; the counts are not.

Table 1: The four delinquency waves of `crim4w`

wave	zeros	missing	maximum	variance / mean
ACRIM	72.0 %	33.4 %	12	3.04
BCRIM	60.6 %	19.2 %	15	3.73
CCRIM	55.0 %	16.6 %	16	3.80
DCRIM	57.0 %	28.1 %	16	3.10

Two properties of Table 1 make these data a useful example throughout: more than half of every wave is zero, and the counts are overdispersed three- to fourfold.

`crim4l` is the same data in long format — 8256 rows for 2064 respondents at four occasions — which is the shape the two-level methods need. There `DELINQ` carries the count, `TIME` the wave and `ID` is the respondent identifier.

The examples of Section 5.7 and Section 5.8 use simulated data instead. A Poisson and a plain negative binomial model need a count *without* an excess of zeros.

5.4 How countimp is organised

This section describes the parts of an imputation call: the engine, the method names, the predictor matrix.

`countimp` version 3 has its own FCS engine. Versions 1 and 2 were simply an extension of `mice` (van Buuren & Groothuis-Oudshoorn, 2011) and could not be used without it.

Why an engine of its own. Being an extension to `mice` meant having to constantly adapt to changes in the `mice` architecture. A change in `mice` could easily break `countimp`. An own imputation engine therefore makes the package more stable and robust. We adapted the FCS sampler and the helpers around it from `mice` under the GPL. `countimp` still integrates well with `mice` and loading `mice` in addition to `countimp` has advantages: Users can call its convergence and plausibility diagnostics on objects created by `countimp`: `plot()`, `densityplot()`, `stripplot()`. Furthermore `quickpred()` can be used to automatically generate imputation models based on the correlation structure of the empirical data. The output of `mice::quickpred()` can be passed to `countimp`. `countimp` furthermore adds diagnostics of its own that `mice` does not have, for the question that count data raise specifically: whether the imputed counts have the right shape:

```
compare.obs.imp(),    compare.percent.count(),    countimp_check(),    countimp_diagnostics(),  
countimp_fit_diag()
```

They are described in Section 5.18.

All in all, `countimp` still stays very close to the `mice` way of doing things: The argument names, the `mice.impute.name` convention for imputation functions and the structure of the result object are those of `mice`, so that anyone who knows the one package can use the other.

The imputation call. Imputations are created by `countimp()`. `m` sets the number of imputations (default `m = 5`), `maxit` the number of iterations of the sampler. Examples in this manual keep those defaults wherever they report anything about the imputed values. Where a call is shown only to see which method or which `predictorMatrix` it produces, `m` and `maxit` are set to 1 or 2 so the manual builds quickly. The model is then specified in one of two ways, set side by side in Section 5.5: through `method` and `predictorMatrix` as in `mice`, or through `formulas` and `family`.

Imputation methods are chosen through the `method` argument, a character vector with one entry per variable in the data set. An empty entry `"` means the variable is complete and needs no imputation. Each name `x` refers to a function `mice.impute.x`; specifying `"nb"` calls `mice.impute.nb()`. The count families are listed in Section 2.3. For the variables that are *not* counts, `countimp` supplies its own versions of the standard methods, so that a mixed data set can be imputed without `mice` installed (Table 2):

Table 2: Non-count imputation methods supplied by `countimp` itself.

method	description	scale
logreg	logistic regression	factor, 2 levels
mean	unconditional mean	numeric
norm	Bayesian linear regression	numeric
norm.nob	linear regression, no parameter uncertainty	numeric
norm.predict	linear regression, predicted value	numeric
pmm	predictive mean matching	numeric
polr	proportional odds model	ordered factor
polyreg	polytomous logistic regression	factor, > 2 levels
sample	random draw from the observed values	any

These are re-implementations: they take the same arguments as their `mice` counterparts and are looked up first, before the `mice` versions. Which set is used can be switched with `options(countimp.methods = "mice")`; the default is `"countimp"`. The two sets differ in one respect: `countimp`'s versions draw the model parameters from their approximate posterior before imputing, where `mice`'s ordinal and multinomial methods condition on the maximum-likelihood fit. Drawing the parameters carries their uncertainty into the between-imputation variance; conditioning on the fit is the simpler route and yields less variable imputations. Neither is the obviously right choice for every data set — `mice` draws the parameters in `norm` and `logreg` and conditions on the fit in the ordinal methods — and `countimp` defaults to the draw so that its non-count methods behave like its count methods. Set the option to `"mice"` when you need to reproduce results that were obtained with `mice`.

One caveat for `polr`. Where the proportional-odds model cannot be fitted — typically because some categories hold only one or two cases — the method falls back to a multinomial model *without raising an error*. Both packages do this, and each records it in its own place: `mice` writes `"polr falls back to multinom"` to the `loggedEvents` of the `mids` object, while `countimp` calls its own `polyreg` and notes `"proportional-odds fit failed: fell back to multinom"` in the diagnostics log that `countimp_diagnostics()` reads. A fallback is not an error and will not appear in a convergence count, so if you rely on `polr` for a variable with many sparsely occupied categories, look for it deliberately after a `countimp()` run in `countimp_diagnostics()`, or after a `mice()` call in `loggedEvents`.

Imputation models are defined through the `predictorMatrix`, a square matrix with one row and one column per variable. Row i gives the imputation model for variable i : a 1 in column j means variable j is used as a predictor, a 0 that it is not. If no matrix is supplied, `countimp()` builds one. The default is that every other variable predicts every incomplete one, with a zero diagonal:

```
library(countimp)
data(crim4w)
imp <- countimp(crim4w[, c("BCRIM", "FEMALE", "RE", "GY")],
               m = 1, maxit = 1, printFlag = FALSE)
imp$predictorMatrix
```

```
      BCRIM FEMALE RE GY
BCRIM      0      1  1  1
```

FEMALE	0	0	0	0
RE	0	0	0	0
GY	0	0	0	0

Two more codes matter for clustered data: -2 marks the grouping variable, and 2 requests a random slope.

From the `predictorMatrix`, `countimp` assembles the three objects that every imputation function receives: `y`, the incomplete variable; `x`, the matrix of predictors selected in that row; and `ry`, the response indicator of `y`, which is `TRUE` where a value was observed. An imputation function returns the values for the `FALSE` positions. Supplying your own imputation functions is possible, if they follow the `mice.impute.mymname(y, ry, x, ...)` style; see `help("mice.impute.nb")` for a worked signature.

5.5 Two ways to specify the model

The model can be specified in either of two ways. They are equivalent for single-level data and differ for clustered data, where the formula route saves a step.

The classical route sets `method` and `predictorMatrix`, as described in Section 5.4. The method name carries the distribution, and the matrix carries the predictors. In the example below, we fit a two-level negative binomial model. The data set for this illustration is `crim4l` (Section 5.3):

```
data(crim4l)
set.seed(20260907)
pm <- matrix(0, ncol(crim4l), ncol(crim4l),
             dimnames = list(names(crim4l), names(crim4l)))
pm["DELINQ", c("FEMALE", "RE", "GY", "TIME")] <- 1
pm["DELINQ", "ID"] <- -2 # -2 marks the grouping variable
meth <- setNames(rep("", ncol(crim4l)), names(crim4l))
meth["DELINQ"] <- "2l.nb"
imp_k <- countimp(crim4l, method = meth, predictorMatrix = pm,
                 m = 2, maxit = 2, printFlag = FALSE)
imp_k$method[imp_k$method != ""]
```

```
DELINQ
"2l.nb"
```

The formula route states the same model as a formula in `lme4` notation and the distribution as a family function. The method name is then derived from both:

```
set.seed(20260907)
imp_f <- countimp(crim4l,
                 formulas = list(DELINQ ~ FEMALE + RE + GY + TIME + (1 | ID)),
                 family = nb(),
                 m = 2, maxit = 2, printFlag = FALSE)
imp_f$method[imp_f$method != ""]
```

```
DELINQ
"2l.nb"
```

Both calls select `2l.nb`. The difference is where the knowledge sits: in the first, you have to know that a -2 in the matrix means “grouping variable” and that the two-level method is called `2l.nb`; in the second, the term `(1 | ID)` is all that needs to be specified. The method name is automatically looked up internally. Note that the family is *not* optional on the formula route. A formula without `family` stops with a message that lists the family constructors:

```
`formulas` needs a matching `family`. Use one of the family
constructors, e.g. family = poisson_count(), nb(), zi_poisson() ...
```

There is no sensible default for count data: the choice between Poisson, negative binomial and the zero-inflated variants is the modelling decision users need to make themselves.

Why `poisson_count()` and not `poisson()`. We have chosen the name `poisson_count()` because the name `poisson()` is already taken. `stats::poisson()` is the family object behind every `glm()` call, and a constructor of the same name here would mask it: after `library(countimp)`, a `glm(y ~ x, family =`

`poisson()` further down the same script would silently be handed the wrong object. Only this one family is affected — `nb()`, `zi_poisson()`, `hurdle_nb()` and `compois()` clash with nothing in `stats`. Trying to set `family = poisson()` results in a message:

```
`family = poisson()` is the family object of stats/glm, not a countimp family.  
Use `family = poisson_count()` instead.
```

Furthermore, also trying to set `family = quasipoisson()` results in a message, because we have deliberately implemented no `quasipoisson` family:

```
`family = quasipoisson()` is the family object of stats/glm, not a countimp family.  
countimp has no quasipoisson family on purpose: quasi-likelihood has no  
posterior to draw from. The method is still available through the classic  
interface: `method = c(<variable> = "quasipoisson")`.
```

Section 5.8.1 describes what the method draws its imputations from once it is reached the classical way.

Note that also `mice` has had a `formulas` argument since version 3.0.0, and it is worth being precise about the difference. There, a formula replaces the `predictorMatrix`: it selects predictors, and the method is still set separately through `method`. A grouping term in the formula does not select a multilevel method — `mice`'s own two-level methods want the grouping variable coded in the predictor matrix. Here, the same formula does both: the grouping term is translated into the level code, and the family plus the term select the method.

What the formula may contain. They may contain plain variable names, a `.` (standing for *every other column*), and one grouping term per level. That is the whole vocabulary. The zero part of a two-part family is *not* written into this formula; it is a formula of its own, passed as `zero`, and the bar means grouping rather than the separator `pscl` uses (Section 5.9). A formula is not a model formula in the `glm()` sense: it is translated into type codes, and a type code names a column. Anything that is not a column of the data therefore has nowhere to go. Transformations and interactions are refused:

```
Variable(s) not in `data`: I(x1^2)
```

The same message follows `x1 * x2`, `x1:x2`, `log(x1)` and `poly(x1, 2)`. To impute under such a model, compute the term as a column of the data first and name that column in the formula — and bear in mind that a squared term or a product imputed as if it were a separate variable is not the same model, which is a question of imputation theory rather than of this interface.

One piece of `glm()` notation has no effect: `offset()` leaves the imputations bit for bit unchanged, so do not read its acceptance as support. The same goes for a `- 1` or `0 +` at the top level of the formula, which does not drop the fixed intercept. Little is lost by that. Written for the cell-means parameterisation of a factor, it describes the same model and changes nothing the imputations depend on; meant literally, it asserts that exactly one event is expected once every predictor is zero, which is a constraint rather than a saving — and an imputation model should be the richer of the two models, not the more constrained. Inside a grouping term, the `0` *does* drop the intercept: `(0 + x1 | id)` works and fits a random slope model with a fixed intercept. That is how the `.noint` variants are selected (Section 5.15).

Exposure time needs no `offset()`. Where counts arise over unequal observation windows, the analysis model usually carries $\log t$ as an offset. The imputation model needs that information too — without it a case observed briefly is imputed at the same rate as one observed for years. Put $\log t$ into the data as a column of its own and name it among the predictors:

```
d$log_t <- log(d$exposure)  
countimp(d, formulas = list(y ~ x1 + log_t), family = poisson_count())
```

This is the more general model. An offset is a predictor whose coefficient is fixed at one; naming the column instead estimates that coefficient, so the imputation model is richer than the analysis model.

5.6 The imputation methods at a glance

Section 2.3 says which family fits which kind of variable. This section lists the `method` names that go with them, and which variants of each exist (Table 3).

Each family has a section of its own with a worked call: Section 5.7 for Poisson counts, Section 5.8 for overdispersion, Section 5.9 for excess zeros, Section 5.10 for hurdle models, Section 5.11 for counts that cannot

be zero, Section 5.12 for underdispersion, Section 5.13 for right-censored counts and Section 5.14 for counts with a known upper bound. The two-level versions follow in Section 5.15 onwards.

Table 3: The 13 count families, their `method` names and their two-level counterparts.

family	method	two-level
<code>poisson_count()</code>	<code>poisson</code>	<code>2l.poisson</code>
<code>nb()</code>	<code>nb</code>	<code>2l.nb</code>
<code>compois()</code>	<code>cmp</code>	<code>2l.cmp</code>
<code>zi_poisson()</code>	<code>zip</code>	<code>2l.zip</code>
<code>zi_nb()</code>	<code>zinb</code>	<code>2l.zinb</code>
<code>hurdle_poisson()</code>	<code>hp</code>	<code>2l.hp</code>
<code>hurdle_nb()</code>	<code>hnb</code>	<code>2l.hnb</code>
<code>zerotrunc_poisson()</code>	<code>ztp</code>	—
<code>zerotrunc_nb()</code>	<code>ztnb</code>	—
<code>censored_poisson()</code>	<code>cp</code>	—
<code>censored_nb()</code>	<code>cnb</code>	—
<code>bounded_poisson()</code>	<code>bp</code>	—
<code>bounded_nb()</code>	<code>bnb</code>	—

Seven of the thirteen families have a two-level counterpart; for the other six (zero-truncated, censored and bounded) only single-level methods exist so far. The reason differs between them.

Those seven are not limited to two levels. A second grouping variable makes the same method three-level, and no further method name is involved — the levels travel in the type codes. Section 5.15.2 works that through for `2l.poisson`, and it stands for all of them; the sections below therefore show the two-level call only.

For censored and bounded counts it is where the likelihood comes from. The families that support multiple levels are fitted with `glmmTMB`, which carries the random effects; the censored and the bounded likelihood are `countimp`'s own. A censored case contributes the tail probability $P(Y \geq c)$, a bounded one is renormalised over its range, and both are maximised directly, with no random-effect layer underneath them. A two-level version is therefore a new estimator, not simply a new argument. This will be an avenue for future program versions.

For zero-truncated counts, implementation is more straightforward. `glmmTMB` supplies `truncated_poisson` and `truncated_nbinom2`, and this package already fits them: the count part of the two-level hurdle methods *is* a two-level zero-truncated model, estimated on the units that passed the gate (Section 5.17). What is missing is a method that uses it on its own, for a variable that is positive by construction. Again, we will implement this in a future program version.

Every family furthermore has at least one bootstrap variant, named with a `.boot` suffix; Section 2.2 explains the distinction between the Bayesian and the bootstrap draw.

In the upcoming sections.

5.7 Imputation of incomplete Poisson distributed variables

The function to impute an incomplete Poisson distributed variable is called `mice.impute.poisson()`. Appending `.boot` to the function name calls the bootstrap regression variant (see `help("mice.impute.poisson.boot")`). The functions have the following arguments:

```
library(countimp)
args("mice.impute.poisson")

function (y, ry, x, wy = NULL, EV = FALSE, ...)
NULL
```

You do not supply `y`, `ry` and `x` yourself: the engine assembles them from the data and the `predictorMatrix` and hands them to the imputation function, as Section 5.4 describes. `wy` is a logical vector of `length(y)` marking the positions for which imputations are to be generated. It defaults to `!ry` — the missing values — and is worth setting only when you want imputations for some other subset.

Argument `EV` defines how the function handles extreme imputations. When set to `TRUE`, the function automatically detects extreme values among the imputations and replaces them by values obtained by predictive mean matching. **This is deprecated.** Replacing the largest draws truncates the upper tail and biases the estimates downwards. `countimp_diagnostics()` took its place in version 3: it flags implausible draws and reports them, deliberately keeping them, because discarding large draws until a small one arrives pulls the imputations downwards (for further details, see Section 5.8).

The Poisson model is fitted to the observed part of the data, `x[ry, ]` and `y[ry]`, with `stats::glm.fit()` from package `stats`. What happens next is what separates the two variants. `mice.impute.poisson()` draws the coefficients from their approximate posterior, $\beta^* \sim N(\hat{\beta}, V(\hat{\beta}))$; `mice.impute.poisson.boot()` instead refits the model on a bootstrap sample of the observed cases and uses those coefficients. Either way the imputations are then drawn from a Poisson distribution with mean $\mu = \exp(x\beta^*)$, the value predicted for the respective incomplete case. Section 2.2 sets the two draws side by side.

5.7.1 Example

We generate an example data set with a Poisson distributed dependent variable y and three continuous predictors x_1 – x_3 and introduce MAR missingness in the dependent variable.

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

## introduce MAR missingness to simulated data
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)
}
MPOIS <- generate.md( POIS, pmis = .2, strength = c( .2, .8 ) )
```

This procedure generates 20% missing data in y . Missingness depends on variable x_1 . Cases whose x_1 -value is above the mean of x_1 are more likely to have a missing value in y . The resulting incomplete data set is called `MPOIS`.

Imputations are then generated by `countimp()`. The formula route states the model and lets the package choose the method:

```
imp <- countimp(MPOIS, formulas = list(y ~ x1 + x2 + x3),
               family = poisson_count(), print = FALSE)
impb <- countimp(MPOIS, formulas = list(y ~ x1 + x2 + x3),
                family = poisson_count(), draw = "boot", print = FALSE)
c(bayes = imp$method[["y"]], boot = impb$method[["y"]])
```

```
      bayes      boot
"poisson" "poisson.boot"
```

`draw = "boot"` selects the bootstrap variant; the default is `"bayes"`.

The same model through `method`. The classical route names the method instead of the model, so you have to know that the Poisson method is called `"poisson"` — and where in the `method` vector it belongs:

```
colnames(MPOIS)
```

```
[1] "y" "x1" "x2" "x3"
```

MPOIS has four variables, so `method` is a vector of length four. `y` comes first, so its entry is `"poisson"` (or `"poisson.boot"`); the three complete variables get `" "`. If `predictorMatrix` is not specified, every other variable predicts the incomplete one:

```
imp_classic <- countimp(MPOIS, method = c("poisson", "", "", ""),
                        print = FALSE)
identical(imp_classic$predictorMatrix["y", ], imp$predictorMatrix["y", ])
```

```
[1] TRUE
```

The resulting object is of class `countimp_mids`, which inherits from `mids`. Analysing the completed data sets and pooling the results is not part of the imputation step, and `countimp` keeps the two separate. `with()` fits the analysis model on every completed data set and returns them together; `miinference()` applies Rubin's rules to that result. Where you want a single completed data set instead, `countimp_complete(imp, k)` returns the k -th. Note that `mice::pool()` works on the same object and gives identical numbers — estimates, standard errors and degrees of freedom agree to the last digit. `miinference()` is `countimp`'s pooling function; `help("miinference")` describes the slots of its output.

```
## Fit the analysis model on each completed data set ...
fits <- with(imp, glm(y ~ x1 + x2 + x3, family = poisson))

## ... and pool. Passing the fits themselves lets miinference() read the
## standard errors and the complete-data degrees of freedom off the models.
pooled <- miinference(fits)
tab <- round(data.frame(estimate = unlist(pooled$est), se = unlist(pooled$std.err),
                        t = unlist(pooled$t.value), df = unlist(pooled$df),
                        fmi = unlist(pooled$fminf)), 4)

## p in the reporting style of the field: three decimals, no leading zero,
## and everything below .001 as such. The exact size of a very small p is
## rarely what a reader needs, and rounding it to 0 states something false.
pval <- unlist(pooled$p.value)
tab$p <- ifelse(pval < .001, "< .001", sub("^0", "", sprintf("%.3f", pval)))
tab[, c("estimate", "se", "t", "df", "p", "fmi")]
```

	estimate	se	t	df	p	fmi
(Intercept)	0.9782	0.0107	91.1378	91.7823	< .001	0.2231
x1	0.7563	0.0085	88.7860	50.7946	< .001	0.3055
x2	-0.2347	0.0089	-26.4806	32.9681	< .001	0.3829
x3	0.4964	0.0096	51.6200	18.0910	< .001	0.5190

Future program versions will get a `summary`-function that produces nicely formatted output.

5.8 Imputation of overdispersed count data

`countimp` has two ways to impute overdispersed counts, each available in a Bayesian and a bootstrap variant:

- 1) imputation based on a quasi-Poisson model, and
- 2) imputation based on a negative binomial model.

Both methods can be used when the equidispersion assumption of the standard Poisson model is violated, i.e. when the variance of the count variable is larger in comparison to its mean.

5.8.1 Imputation based on a quasi-Poisson model

Method "quasipoisson" imputes univariate missing data based on a Bayesian quasi-Poisson GLM, fitted with the standard R function `glm.fit` and `family = quasipoisson(link = log)`. "quasipoisson.boot" is the bootstrap variant; "qpois" and "qpois.boot" are aliases and behave identically.

A quasi-likelihood is not a distribution, so there is nothing to draw the imputations from directly. The fit supplies a mean function together with the Pearson dispersion ϕ , for which $\text{Var}(Y) = \phi\mu$, and the imputations are drawn from the negative binomial matched to exactly that mean and variance:

$$\text{size} = \frac{\mu}{\phi - 1}, \quad \text{Var}(Y) = \mu + \frac{\mu^2}{\text{size}} = \phi\mu.$$

The identity requires $\phi > 1$. Where the fitted dispersion is at most 1, the counts are equi- or underdispersed, which happens whenever a count has an upper bound, and can happen by sampling alone. In that case no negative binomial can represent $\text{Var} < \mu$, so the imputations are drawn from a Poisson instead and a warning says so once per session. Poisson draws overstate their variance rather than understate it, and so err towards intervals that are too wide. For a variable that is underdispersed by design, prefer `cmp` (Section 5.12), which explicitly models underdispersion.

ϕ enters the Bayesian variant a second time. The coefficients are drawn from $\mathcal{N}(\hat{\beta}, \phi V_{\text{unscaled}}(\hat{\beta}))$, i.e. from the *scaled* covariance, which is the sampling variance of $\hat{\beta}$ under a quasi-likelihood with estimated dispersion.

Two limits of the method are deliberate. There is no `quasipoisson` family constructor, so the formula route of Section 5.5 does not reach it: a quasi-likelihood has no likelihood and hence no posterior to draw from, and writing `family = quasipoisson()` reports this rather than failing obscurely. And there is no two-level quasi-Poisson — for clustered overdispersed counts use `2l.nb`.

5.8.2 Example

We simulate an overdispersed count variable `y` and generate about 20% missing data in this variable. Imputation, repeated data analysis and pooling of results works analogous to Section 5.7.1.

```
## simulate overdispersed count data
set.seed( 1234 )
b0 <- 1
b1 <- .75
b2 <- -.25
b3 <- .5
N <- 5000
x1 <- rnorm(N)
x2 <- rnorm(N)
x3 <- rnorm(N)
mu <- exp( b0 + b1 * x1 + b2 * x2 + b3 * x3 )
y <- MASS::rnegbin( N, theta = 2, mu )
NB <- data.frame( y, x1, x2, x3 )

## introduce MAR missingness to simulated data
total <- round( .2 * N ) ##number of missing data in y
sm <- which( NB[,2] < mean( NB[,2] ) ) ##subset: cases with x1<mean(x1)
gr <- which( NB[,2] > mean( NB[,2] ) ) ##subset: cases with x1>mean(x1)
sel.sm <- sample( sm, round( .2 * total ) ) ##select cases to set as missing
sel.gr <- sample( gr, round( .8 * total ) ) ##select cases to set as missing
sel <- c( sel.sm, sel.gr )
MNB <- NB
MNB[sel,1] <- NA ##delete selected data

## impute missing data
imp <- countimp( MNB, method = c( "quasipoisson", "", "", "" ), print = FALSE)

## repeated data analysis and pooling of results
```

```
fits <- with(imp, glm(y ~ x1 + x2 + x3, family = quasipoisson))
pooled <- miinference(fits)
tab <- round(data.frame(estimate = unlist(pooled$est), se = unlist(pooled$std.err),
                        t = unlist(pooled$t.value), df = unlist(pooled$df),
                        fmi = unlist(pooled$fminf)), 4)
pval <- unlist(pooled$p.value)
tab$p <- ifelse(pval < .001, "< .001", sub("^0", "", sprintf("%.3f", pval)))
tab[, c("estimate", "se", "t", "df", "p", "fmi")]
```

	estimate	se	t	df	p	fmi
(Intercept)	1.0123	0.0177	57.0782	92.5262	< .001	0.2221
x1	0.7400	0.0182	40.6153	12.7943	< .001	0.6136
x2	-0.2405	0.0135	-17.8236	90.1138	< .001	0.2253
x3	0.4668	0.0130	35.8500	114.0088	< .001	0.1986

5.8.3 Imputation based on a negative binomial model

Method "nb" fits a negative binomial model using function `glm.nb()` from package MASS and simulates the imputations from a negative binomial distribution, using function `rnegbin()` from package MASS.

The imputation process works analogous to Section 5.7.1.

Why we have deprecated `EV = TRUE`. The outlier handling described in Section 5.7 replaced extreme imputations by predictive mean matching draws. For an overdispersed count the large values are not necessarily artefacts, so removing them could remove part of the target distribution. A test simulation of the package measured its effects. The data in this test were negative binomial with $\theta = 2$, which at the coefficients used gave a variance-to-mean ratio of about 3, at $n = 500$, with missingness depending on one of the covariates, over 1000 replications. With `EV = TRUE` the relative bias was -14.3 percent and coverage of the nominal 95 percent interval 79.2 percent, against -0.3 percent and 94.1 percent with `EV = FALSE`. The old mechanism could not tell a heavy tail from a pathological fit. On clean negative binomial data it flagged values in 98 percent of all draws, which is why `countimp_diagnostics()` now reports instead of repairing automatically.

5.9 Imputation of count data with an excess number of zero counts

Methods `zip` and `zinb` impute missing data based on a zero-inflated Poisson or NB model respectively (for details, see Kleinke & Reinecke, 2013). Methods `hp` and `hnb` use a hurdle Poisson or NB model. All four are fitted with `pscl`, which has to be installed separately.

Which of the four. The two families differ in where the zeros come from. Under zero inflation (Lambert, 1992) a zero can arise from either part — from a structural source, or as a sampled zero of the count process. Under a hurdle model (Mullahy, 1986) the zeros come from the zero part alone, and the count part is zero-truncated over the positive values. The choice depends on the theory and the data: is there a subgroup that could not have produced a count at all, or does everyone face the same threshold and differ only in how often they clear it? The second choice is the easier one — `zip` and `hp` take a Poisson count part, `zinb` and `hnb` a negative binomial one, so the choice depends on whether the positive counts are overdispersed.

Users can specify different sets of predictors for the zero and count parts of the models. Allowed entries in the `predictorMatrix` are listed in Table 4.

Table 4: Allowed entries in the `predictorMatrix`

numeric code	meaning
0	variable is not included in either model
1	variable is included in both the zero and the count model
2	count model predictor
3	zero-model predictor

The same model through the formula interface. Instead of coding the two parts numerically, the predictors are specified separately via two arguments: `formulas` carries the count part, `zero` the inflation part:

```
## count model: x1 and x2; zero model: z1
countimp(data, formulas = list(y ~ x1 + x2), zero = ~ z1,
          family = zi_poisson())
```

This is the equivalent of $x_1 = 2$, $x_2 = 2$, $z_1 = 3$ in Table 4. Omitting `zero` reuses the count part's right-hand side, which puts every predictor into both parts — the equivalent of setting all entries to 1, and the shorter form whenever the two parts share their predictors.

`zero` accepts more than the one-sided form above. A two-sided formula (`zero = y ~ z1`) names the imputed variable explicitly, and a named list (`zero = list(y = ~ z1, w = ~ x1)`) is the form to use when several variables are imputed in the same call.

Warning

A syntax trap for users of `pscl`. In `pscl::zeroinfl()` and `pscl::hurdle()` the two model parts are separated by a vertical bar, as in `y ~ x1 + x2 | z1`. Writing that here does **not** produce an error — and it does not produce a two-part model either. In our syntax, the bar is the grouping operator of `lme4`, so `countimp` reads the term after it as a **cluster variable** and consequently selects a two-level method:

```
## NOT a zero model with z1 --- this selects 2l.zip and treats z1 as a cluster
countimp(data, formulas = list(y ~ x1 + x2 | z1), family = zi_poisson())
```

Use `zero` for the second model part and reserve the bar for grouping. Because the mistake is silent, checking `imp$method` after a two-part call is worth the one line it takes.

The coding above is `countimp`'s own. `mice`'s `predictorMatrix` admits only 0 and 1 unless the method carries the `.2l` prefix, and that is why Kleinke & Reinecke (2013) labelled the single-level functions `2l.zip` and `2l.zinb`: it was the only channel through which separate predictor sets for the zero and the count part could reach the method. The name confused empirical researchers, because those functions fitted no two-level model. Version 3 reads the codes in its own engine, so the restriction is gone and the names mean what they say.

Caution: methods `2l.zip` and `2l.zinb` now do fit two-level models — see Section 5.17.

5.9.1 Example

The data set for this example is `crim4w` (Section 5.3). We impute its second wave, BCRIM: 19 percent missing, 61 percent of the observed counts zero, a maximum of 15 offences — which is what makes it a candidate for a zero-inflated model. The predictors `FEMALE`, `RE` and `GY` are fully observed.

The call below gives the count part all three predictors and the zero part two of them. This decision is neither theory-driven nor data-driven; the purpose is simply to show that the two parts need not share their predictors, which is what argument `zero` is for and what the codes 2 and 3 of Table 4 express on the classical route.

```
data(crim4w)
dz <- crim4w[, c("BCRIM", "FEMALE", "RE", "GY")]
set.seed(20260907)
imp_zinb <- countimp(dz,
  formulas = list(BCRIM ~ FEMALE + RE + GY),
  zero      = ~ FEMALE + GY,
  family    = zi_nb(),
  m = 5, maxit = 5, printFlag = FALSE)
imp_zinb$method[["BCRIM"]]
```

```
[1] "zinb"
```

```
c(observed = mean(dz$BCRIM == 0, na.rm = TRUE),
  imputed   = mean(unlist(imp_zinb$imp$BCRIM) == 0))
```

```
observed  imputed
0.6064787 0.6327456
```

The second line is the check a user can make without knowing the truth: does the share of zeros among the imputations resemble the share among the observed values? The two need not agree exactly. Wherever the mechanism is not MCAR the missing cases differ systematically from the observed ones, so under MAR a gap is *expected* — but a wide one is worth a second look.

`zi_poisson()` selects `zip` instead, and appending `.boot` to the method name on the classical route selects the bootstrap variant of either. The hurdle counterpart of the same model, where a zero can come from one process only, follows in Section 5.10.

5.9.2 Simulation evidence

The congenial `zip` and `zinb` models were evaluated by Kleinke & Reinecke (2013), who found them to perform adequately well when the data were in fact zero-inflated Poisson or negative binomial. The re-run reported in Kleinke & Reinecke (2026) holds the two against each other on one and the same data: imputing zero-inflated NB data with the leaner `zip` model is damaging, whereas imputing ZIP data with the richer `zinb` model costs essentially nothing. Where the distributional form is uncertain, the more flexible of two nested count models is the safer choice. The hurdle variants of the same functions are evaluated in Section 5.10.

5.10 Imputation of hurdle distributed count data

A hurdle model treats the zeros as a process of their own: a first stage decides whether the count clears zero at all, and a second stage models the positive counts with a zero-truncated distribution (Mullahy, 1986).

The distinction (zero-inflation versus hurdle) is an assumption about the data, so the choice belongs to the substantive question. In the criminology data used here, the question is whether everyone faces the same first-offence hurdle — if so, a hurdle model fits; if some respondents are structurally unable to offend, then zero-inflation is the obvious choice.

`countimp` supplies `hp` for a hurdle Poisson count part and `hnb` for a hurdle negative binomial one. Both are fitted with `pscl::hurdle()`, so package `pscl` has to be installed for them. Both are two-part families, so they take a second formula for the zero component through the `zero` argument. The example imputes BCRIM from `crim4w` again (Section 5.3):

```
data(crim4w)
dh <- crim4w[, c("BCRIM", "FEMALE", "RE", "GY")]
set.seed(20260907)
imp_hp <- countimp(dh,
  formulas = list(BCRIM ~ FEMALE + RE + GY),
  zero      = ~ FEMALE + GY,
  family    = hurdle_poisson(),
  m = 5, maxit = 5, printFlag = FALSE)
imp_hp$method[["BCRIM"]]
```

```
[1] "hp"
```

The count part regresses BCRIM on all three predictors, the zero part on two of them. `zero` also accepts a two-sided formula (`zero = BCRIM ~ FEMALE + GY`) and a named list, which is the form to use when several variables are imputed. Leaving `zero` out is allowed — the zero part then uses the same predictors as the count part.

Whether the imputations reproduce the share of zeros is the diagnostic a user can check without knowing the truth:

```
c(observed = mean(dh$BCRIM == 0, na.rm = TRUE),
  imputed   = mean(unlist(imp_hp$imp$BCRIM) == 0))
```

```
observed  imputed
0.6064787 0.5954660
```

The negative binomial variant is called the same way and adds a dispersion parameter, which matters whenever the positive counts are more variable than a Poisson allows:

```
set.seed(20260907)
imp_hnb <- countimp(dh,
```

```

      formulas = list(BCRIM ~ FEMALE + RE + GY),
      zero      = ~ FEMALE + GY,
      family    = hurdle_nb(),
      m = 5, maxit = 5, printFlag = FALSE)
rbind(hp = c(zeros = mean(unlist(imp_hp$imp$BCRIM) == 0),
            var = var(unlist(imp_hp$imp$BCRIM)),
            max = max(unlist(imp_hp$imp$BCRIM))),
      hnb = c(zeros = mean(unlist(imp_hnb$imp$BCRIM) == 0),
            var = var(unlist(imp_hnb$imp$BCRIM)),
            max = max(unlist(imp_hnb$imp$BCRIM))))

```

	zeros	var	max
hp	0.595466	2.631769	9
hnb	0.595466	3.293614	17

The share of zeros is the same for both families — to the last digit, when the seed is the same. That is not a coincidence and not a bug: in both families the zero part is the same logistic model on the same predictors, so with the same seed it makes the same zero-or-not decisions. Only the count part differs. The choice between `hp` and `hnb` therefore has to be diagnosed on the *positive* counts, where the variance and the largest imputed value separate the two clearly. Reading the share of zeros to compare them would suggest they are interchangeable, which they are not.

Both families have `.boot` variants that replace the Bayesian parameter draw by a bootstrap. All four variants are covered by the evidence below.

5.10.1 Simulation evidence

Kleinke & Reinecke (2013) evaluated the zero-inflation methods; the hurdle variants were added in version 2 and are evaluated on a closely similar design in Kleinke & Reinecke (2026), which is the citable source for it. With a congenial model the regression coefficients of both parts are recovered and coverage stays near the nominal level.

The one result worth carrying into practice concerns the dispersion parameter. Hurdle-NB data imputed with a model that has none — `hp`, `hp.boot` or `zip` — overestimate θ substantially, and the coverage of its interval falls towards zero as the sample grows. The mechanism recurs throughout this manual: an imputation model that ignores a feature of the data does not merely add noise. It introduces a bias that a larger sample cannot dilute but does make more visible, because the confidence intervals shrink around the wrong value.

5.11 Imputation of zero-truncated count data

Some counts cannot be zero by construction like the number of days spent in hospital for admitted patients, the number of offences among convicted offenders, or the number of children among mothers. An ordinary Poisson or negative binomial imputation model *will* impute zeros there. Methods `ztp` and `ztnb` fit the zero-truncated distributions and draw only from the admissible range.

Note that if such a count has a ceiling as well, e.g. at least 1 and at most 8, the bounded family is the one to choose: `bounded_poisson(bounds = c(1, 8))`. See section Section 5.14 for details.

In the example below, we take `crim4w` (Section 5.3) and only look at participants who have committed at least one punishable offence. So, our study population here is the subgroup of offenders. For the sake of this illustration alone, we assume that participants with missing data on BCRIM are offenders.

```

dz <- crim4w[is.na(crim4w$BCRIM) | crim4w$BCRIM > 0,
             c("BCRIM", "FEMALE", "RE", "GY")]
set.seed(20260907)
imp_ztp <- countimp(dz,
                    formulas = list(BCRIM ~ FEMALE + RE + GY),
                    family    = zero trunc poisson(),
                    m = 5, maxit = 5, printFlag = FALSE)
c(n = nrow(dz), missing = sum(is.na(dz$BCRIM)),
  smallest_imputed = min(unlist(imp_ztp$imp$BCRIM)))

```

n	missing	smallest_imputed
1053	397	1

The smallest imputed value is 1, as it must be. Calling `ztp` on data that do contain zeros stops with a message naming the problem rather than silently fitting the wrong model.

`ztnb` adds a dispersion parameter. On these data the positive counts are more variable than a Poisson allows, so it is the better choice:

```
set.seed(20260907)
imp_ztnb <- countimp(dz,
  formulas = list(BCRIM ~ FEMALE + RE + GY),
  family    = zerotrunc_nb(),
  m = 5, maxit = 5, printFlag = FALSE)
c(dispersion_observed = var(dz$BCRIM, na.rm = TRUE) / mean(dz$BCRIM, na.rm = TRUE),
  ztp = var(unlist(imp_ztp$imp$BCRIM)) / mean(unlist(imp_ztp$imp$BCRIM)),
  ztnb = var(unlist(imp_ztnb$imp$BCRIM)) / mean(unlist(imp_ztnb$imp$BCRIM)))
```

dispersion_observed	ztp	ztnb
2.0982482	0.9237434	2.3371105

Note that an observed zero under `ztp` or `ztnb` produces an error, because the value lies outside the model's support.

5.11.1 Simulation evidence

The zero-truncated families are new in version 3, and Kleinke & Reinecke (2026) report their evaluation. Over six designs of 500 replications each, `ztp`, `ztp.boot` and `ztnb` show no bias worth the name (at most a tenth of the estimate's own standard error) and coverage between 92.8 and 96.6 per cent, which is within Monte Carlo error of the nominal level throughout.

5.12 Imputation of underdispersed count data

All the models so far can represent equidispersion or more variance than the mean. Counts with *less* variance than the mean occur too — when a process has an upper limit, or when responses cluster tightly around a typical value. Fitting a Poisson model there overstates the variance and, after imputation, produces confidence intervals that are too wide. The Conway-Maxwell-Poisson distribution covers both directions through a dispersion parameter (Shmueli et al., 2005), and `cmp` imputes with it.

`cmp` accepts neither a `censor` nor a `bounds` argument, and the reason is the normalising constant. The probability mass function is $P(Y = y) = \lambda^y / (y!)^\nu / Z(\lambda, \nu)$ with $Z(\lambda, \nu) = \sum_{j \geq 0} \lambda^j / (j!)^\nu$, an infinite series with no closed form. `countimp` evaluates it by summing a window of terms wide enough for the required accuracy. A censored likelihood needs the tail probability $P(Y \geq c)$ and a bounded one the mass over the admissible range; both mean summing that series over a further range, which would be another estimator. Where the ceiling is itself the source of the underdispersion (the first of the two cases named above) the bounded families should be selected instead: a count confined to a range is underdispersed by construction, and `bounded_poisson()` models the confinement instead of absorbing it into a dispersion parameter.

The example below draws from a binomial distribution because that is underdispersed by construction: its variance is $Kp(1 - p) < Kp = \mu$.

```
set.seed(20260907)
N <- 600
x <- rnorm(N)
du <- data.frame(y = rbinom(N, 8, plogis(-0.3 + 0.5 * x)), x = x)
du$y[sample(N, 120)] <- NA # 20 percent MCAR
imp_cmp <- countimp(du, formulas = list(y ~ x), family = compois(),
  m = 5, maxit = 5, printFlag = FALSE)
c(observed = var(du$y, na.rm = TRUE) / mean(du$y, na.rm = TRUE),
  imputed = var(unlist(imp_cmp$imp$y)) / mean(unlist(imp_cmp$imp$y)))
```

observed	imputed
0.7245367	0.7756709

As we can see, both the observed and the imputed variance-to-mean ratio stay below one.

i Note

When the imputation model does not fit the data: **trying to impute with method `cmp` fails on `crim4w`** (Section 5.3). Fitting it to BCRIM stops with **the coefficient covariance matrix is not positive definite, so no coefficients can be drawn.**

A direct `glmmTMB::compois()` fit to the complete rows converges in about a second. However, the imputation needs more than the fit — it needs the *covariance matrix of the coefficients* to draw parameters from, and on these data that matrix is singular. BCRIM has a variance-to-mean ratio of 3.73, so it is strongly *overdispersed* — the regime `cmp` is least suited to, and where `nb` is obviously the better option. The lesson to learn: a family can be estimable and still not be usable for imputation. `countimp_diagnostics()` reports what most likely went wrong.

5.12.1 Simulation evidence

`cmp` is new in version 3 and its evaluation is reported in Kleinke & Reinecke (2026), over 24 designs of 500 replications each. The result does not flatter the family. The intercept is recovered everywhere, but the slope is covered at 88.2 per cent where the effect is strong, against 96.0 for `bp` — and the reason is not the dispersion but the ceiling: `cmp` does not know the upper bound of the scale and continues the slope past it, overstating it by about six per cent of its true value.

On the dispersion itself, the quantity the family exists for, `pmm` lands at a ratio of 1.003 to the truth and `cmp` at 1.069, so `cmp` is roughly twenty times further off at some three hundred times the computing time. Its coverage of that parameter is the better of the two (91.7 against 90.5 per cent), and both are far above what a Poisson model manages (40.2), but the point estimate is what the family was built to get right.

For practice the more useful result is the one above: the family is the most demanding of the thirteen to fit, and a fit that converges on complete data need not yield a covariance matrix the draw can use. Always check `countimp_diagnostics()` after a `cmp` call rather than assuming it well behaved.

5.13 Imputation of right-censored counts

A count is right-censored when the recording stops at some limit: a questionnaire that offers “x or more” as its top category, such as 10 or more criminal offences. The observed value at the limit means “at least this much”, not “exactly this much”. An imputation model that ignores the censoring will happily impute values above the limit.

`cp` and `cnb` fit censored Poisson and censored negative binomial models. Note that the `crim4w` data (Section 5.3) are not censored — their delinquency score is bounded at 16 by construction (see Section 5.14). To show what the censored families do, the example below top-codes BCRIM at 10, as if the questionnaire had used “10 or more” as its highest category.

The censoring is therefore imposed for the purpose of this example:

```
data(crim4w)
dc <- crim4w[, c("BCRIM", "FEMALE", "RE", "GY")]
dc$BCRIM <- pmin(dc$BCRIM, 10L) # top-code at 10, as a survey would
set.seed(20260907)
imp_cnb <- countimp(dc,
  formulas = list(BCRIM ~ FEMALE + RE + GY),
  family    = censored_nb(censor = 10),
  m = 5, maxit = 5, printFlag = FALSE)
range(unlist(imp_cnb$imp$BCRIM))
```

```
[1] 0 10
```

The imputations stay within the range the data can take. The contrast with a model that does not know about the limit is the point of the family:

```
set.seed(20260907)
imp_nb <- countimp(dc, formulas = list(BCRIM ~ FEMALE + RE + GY),
                  family = nb(), m = 5, maxit = 5, printFlag = FALSE)
rbind(cnb = range(unlist(imp_cnb$imp$BCRIM)),
      nb = range(unlist(imp_nb$imp$BCRIM)),
      observed = range(dc$BCRIM, na.rm = TRUE))
```

	[,1]	[,2]
cnb	0	10
nb	0	26
observed	0	10

The plain negative binomial model imputes up to 26, well above a limit of 10. Three things can go wrong when that happens:

The variable acquires values the instrument could not produce. A questionnaire whose top category is “10 or more” cannot yield an 11, let alone a 26. The plain `nb` model produces 14 such draws out of 1985, taking the values 11, 12, 13, 14, 15, 26. A frequency table of the completed variable therefore lists categories no respondent was ever offered.

A mean is pulled upward. In our example this effect is barely noticeable, because the draws above the limit (second column in the output below) are few:

```
completed <- function(im) apply(seq_len(im$m), function(k) countimp_complete(im, k)$BCRIM)
round(rbind(
  observed = c(mean = mean(dc$BCRIM, na.rm = TRUE),
               p_ge_10 = mean(dc$BCRIM >= 10, na.rm = TRUE),
               max = max(dc$BCRIM, na.rm = TRUE)),
  cnb = c(mean(completed(imp_cnb)), mean(completed(imp_cnb) >= 10),
          max(completed(imp_cnb))),
  nb = c(mean(completed(imp_nb)), mean(completed(imp_nb) >= 10),
          max(completed(imp_nb))), 3)
```

	mean	p_ge_10	max
observed	1.042	0.010	10
cnb	1.052	0.011	10
nb	1.053	0.010	26

A proportion above the threshold has no data behind it. Under censoring, every observation recorded as 10 means “10 or more”, and nothing in the data separates an 11 from a 20. The share *at or above* 10 is therefore estimable and both models agree on it; the share *above* 10 is not estimable at all. The censored model does not pretend otherwise and leaves its imputations at the limit. The plain `nb` model fills the gap by extrapolating its tail, so a sentence of the form “x percent reported more than ten offences” would rest entirely on values the study never measured.

Rounding the imputations to the limit seems an obvious repair to these problems. However, see Section 5.14. We recommend to use the imputation functions that support censoring instead of applying quick fixes to the data imputed under an inappropriate model.

💡 Tip

The family checks the limit against the data. Passing `censor = 10` while the variable still contains larger values stops with 12 observed value(s) exceed their censoring limit, e.g. `y = 12` with `censor = 10`. A censored observation is recorded AT the limit, never above it, so either the limit is wrong or the variable is not censored. which is worth reading as a definition: if values above the limit exist, the variable is not censored at that limit, and `cp/cnb` are the wrong family.

5.13.1 Simulation evidence

Kleinke & Reinecke (2026) evaluate the censored families over 24 designs of 500 replications each. On the manifest (censored) scale the coefficients come out covered close to the nominal level throughout: 96.0 to 96.6 per cent for `cnb` and 94.0 to 94.9 for `cp`, across censoring rates of 5, 15 and 30 per cent, with no bias worth reporting.

5.14 Imputation of counts with a known upper bound

A bound is not the same thing as a censoring limit. A censored observation at 10 means “10 or more” — the true value may be larger and is unobserved. A bounded count of 10 out of 10 means exactly ten, and eleven is impossible: e.g. the number of correct answers in a ten-item test. `bp` and `bnb` keep the imputations inside the admissible range.

BCRIM of `crim4w` (Section 5.3) is such a count. The CrimoC instrument asked about sixteen classes of punishable offence, and the delinquency score counts how many of them a respondent reported. It is thus a diversity index: its upper bound is 16 by construction, and seventeen is not a rare value but an impossible one. That the observed maximum of this wave is 15 is a property of the sample, not of the scale.

```
db <- crim4w[, c("BCRIM", "FEMALE", "RE", "GY")]
set.seed(20260907)
imp_bnb <- countimp(db,
  formulas = list(BCRIM ~ FEMALE + RE + GY),
  family    = bounded_nb(bounds = c(0, 16)),
  m = 5, maxit = 5, printFlag = FALSE)
c(
  observed_max = max(db$BCRIM, na.rm = TRUE),
  imputed_max  = max(unlist(imp_bnb$imp$BCRIM)),
  bound        = 16)
```

observed_max	imputed_max	bound
15	16	16

Why not simply round the imputations afterwards? It is an obvious thought and the post-processing tools of either `mice` or `countimp` would make it simple to do so: let a plain model impute, then replace everything above the bound by the bound. Kleinke & Reinecke (2026) puts that procedure to the test and compares its performance against the bounded families directly. Averaged over 42 design cells, the regression coefficients are covered in 95.5 percent of replications under `bp` and only in 86.2 percent when a Poisson model imputes and its draws are rounded afterwards. `pmm` is between the two at 93.9 percent. It respects the bound without having to be told that the data are bounded, because it copies observed values and can’t impute a value that is not present in the observed data (Section 4.1).

For completeness, the machinery for doing it anyway is there. `countimp()` accepts `mice`’s `post` argument and applies it in the same way: a character vector with one expression per variable, evaluated after that variable has been imputed, with the draws reachable as `imp[[j]][, i]`.

```
post <- rep("", ncol(d)); names(post) <- names(d)
post["BCRIM"] <- "imp[[j]][, i] <- pmin(imp[[j]][, i], 16)"
```

The same expression works through `mice()` itself, which finds this package’s methods by their names. One detail if you go that way: `mice::squeeze()`, the usual idiom in `mice`’s own examples, needs its `mice::` prefix here unless `mice` is attached, because the expression is evaluated where `countimp` can see it and `mice` is only a suggested package and not loaded by default.

Note that the lower bound need not be fixed at zero. `bounds = c(1, 8)` for example describes a count that can be neither zero nor larger than eight, a scale with a floor as well as a ceiling.

Choosing between the censored and bounded families follows the same logic as everywhere in this manual: think about what the variable is, not which model converges. If the top category of your questionnaire reads “15 or more”, the variable is censored and `censored_nb(censor = 15)` is the right choice. If it reads “15” and fifteen is the maximum possible, the variable is bounded and `bounded_nb(bounds = c(0, 15))` is right. The two families give different imputations at the top of the range, and the difference is the respective model assumption.

5.14.1 Simulation evidence

The bounded families are new in version 3, and Kleinke & Reinecke (2026) evaluate them over 42 designs of 500 replications each. The bounded methods produce accurate confidence interval coverage. Point estimates are widely accurate.

5.15 Imputation of two-level Poisson distributed data

Method `2l.poisson` imputes missing data based on a generalized linear mixed effects Poisson model. Note that the function requires the data to be in long format, i.e. the repeated measurements are stacked upon another. Additional time and id variables indicate measurement timepoint (when the data are panel data) and cluster membership. The function is described in detail in Kleinke & Reinecke (2015a). The `predictorMatrix` codes below are identical to the ones `mice` uses for its two-level methods. Allowed entries are given in Table 5:

Table 5: Allowed entries in the `predictorMatrix`

numeric code	meaning
0	variable is not included in the imputation model
1	variable is included as a fixed effect
2	variable is included as a fixed <i>and</i> random effect
-2	class variable

The same model through the formula interface. The numeric codes translate one-to-one into the notation `lme4` and `glmmTMB` users already know: a predictor listed in the fixed part is a 1, one inside the random-effects term is a 2, and the grouping factor after the `|` replaces the -2.

```
## fixed effect for x1, fixed and random for x2, id as the class variable
countimp(data, formulas = list(y ~ x1 + (1 + x2 | id)),
          family = poisson_count())
```

Both routes produce the same `predictorMatrix`.

Dropping the random intercept. By default the intercept is both a fixed and a random effect. The `.noint` variants keep it fixed and take it out of the random part (see `help("mice.impute.2l.poisson")`). On the formula route we use the ordinary `lme4` spelling:

```
## random slope for TIME, no random intercept --- selects 2l.poisson.noint
countimp(crim4l, formulas = list(DELINQ ~ FEMALE + TIME + (0 + TIME | ID)),
          family = poisson_count())
```

0 or -1 treat the intercept as fixed. `(-1 + TIME | ID)` works, as well.

This is one place where the two routes are *not* interchangeable. The `predictorMatrix` of the two calls is identical — `TIME` carries a 2, `ID` a -2, either way — because the matrix has no code for “random slope but no random intercept”. On the classical route the distinction therefore lives in the method name alone, and you have to write `method = c(DELINQ = "2l.poisson.noint")` yourself.

5.15.1 Example

The data are `crim4l` (Section 5.3), the long-format shape the two-level methods need: `ID` marks the person, `TIME` the wave, and `DELINQ` is the incomplete count.

```
data(crim4l)
set.seed(20260907)
imp_2lp <- countimp(crim4l,
                    formulas = list(DELINQ ~ FEMALE + RE + GY + TIME + (1 | ID)),
                    family = poisson_count(),
                    m = 5, maxit = 5, printFlag = FALSE)
imp_2lp$method[["DELINQ"]]
```

```
[1] "2l.poisson"
```

The formula was translated into the codes of Table 5, which is what the classical route would have you write by hand:

```
imp_2lp$predictorMatrix["DELINQ", ]
```

FEMALE	RE	GY	TIME	DELINQ	ID
1	1	1	1	0	-2

ID carries the -2 that marks the class variable, the four predictors carry a 1, and the imputed variable itself a 0. A random slope for TIME would be a 2 there, and (1 + TIME | ID) on the formula route.

5.15.2 Three levels

Everything in this section and the ones that follow extends to a third level, and the extension needs no new method name. This example stands for all of them: what it shows for 2l.poisson holds for 2l.nb, 2l.zip, 2l.hnb, 2l.cmp and their variants alike.

The data are simulated — pupils in classes in schools, a count outcome with a random intercept at each of the two grouping levels, and a MAR mechanism on the single predictor:

```
set.seed(20260927)
n_s <- 12; n_c <- 4; n_p <- 15 # schools, classes per school, pupils per class
school <- rep(seq_len(n_s), each = n_c * n_p)
class <- rep(seq_len(n_s * n_c), each = n_p) # numbered across schools
u_s <- rnorm(n_s, 0, 0.4)[school]
u_c <- rnorm(n_s * n_c, 0, 0.3)[class]
x1 <- rnorm(length(school))
y <- rpois(length(school), exp(0.8 + 0.5 * x1 + u_s + u_c))
d <- data.frame(y, x1, school, class) # outer level first, as the formula writes it
d$y[rbinom(nrow(d), 1, plogis(-1.2 + 0.8 * x1)) == 1] <- NA
c(n = nrow(d), schools = n_s, classes = n_s * n_c,
  percent_missing = round(100 * mean(is.na(d$y)), 1))
```

n	schools	classes	percent_missing
720.0	12.0	48.0	27.8

A second grouping term is all that distinguishes the call from the two-level one above:

```
imp_3l <- countimp(d,
  formulas = list(y ~ x1 + (1 | school) + (1 | class)),
  family = poisson_count(),
  m = 5, maxit = 5, printFlag = FALSE)
imp_3l$method[["y"]]
```

```
[1] "2l.poisson"
```

```
imp_3l$predictorMatrix["y", ]
```

y	x1	school	class
0	1	-2	-2

The method is still 2l.poisson, and the name is inherited rather than descriptive. It comes from mice, whose own 2l.norm, 2l.pan and 2l.lmer do mean two levels and stop at them. This package kept the names when it extended the methods to a third level, and that is a deliberate trade: the names are what the engine dispatches on, they have been the public interface since version 2, and Kleinke et al. (2020) prints them throughout. A more accurate label such as ml.poisson would cost every existing script and every printed reference without changing what the method does. The levels travel in the type codes, not in the name: when the model is a three-level-model, **two columns carry a -2**. One grouping variable makes the model two-level, a second makes it three-level, and that is the range the package documents and tests. Future program versions will support more than three levels.

Specifying the imputation model via the classical route simply requires setting the second -2 by hand. It gives the same imputations from the same seed as the formula specification:

```

pm <- matrix(0, ncol(d), ncol(d), dimnames = list(names(d), names(d)))
pm["y", "x1"] <- 1
pm["y", c("school", "class")] <- -2

set.seed(11)
a <- countimp(d, method = c(y = "2l.poisson", x1 = "", class = "", school = ""),
              predictorMatrix = pm, m = 2, maxit = 2, printFlag = FALSE)
set.seed(11)
b <- countimp(d, formulas = list(y ~ x1 + (1 | school) + (1 | class)),
              family = poisson_count(), m = 2, maxit = 2, printFlag = FALSE)
identical(unlist(a$imp$y), unlist(b$imp$y))

```

[1] TRUE

Two rules come with the extra level. First, **random slopes stay with the first grouping term**. The reason is the channel the model travels through: a method receives one row of type codes and a method name, and that row has room for one slope set and one intercept flag. A 2 says *random slope*, but there is no second code to say *at which level*, so with more than one grouping variable the level has to be fixed by convention. $(1 + x1 | \text{school}) + (1 | \text{class})$ is accepted; putting the slope on the second term is refused:

Only the first random-effect term may carry random slopes or drop its intercept.

Which level carries the slopes is decided by the order of the grouping columns in the data, not by the order of the terms in the formula. The data above put `school` before `class`, so a random slope has to attach to `school`: the random part needs to be built as $(1 + x1 | \text{school}) + (1 | \text{class})$. To attach the random slopes to the class-level instead, move that column in front of the other grouping column:

```
d <- d[, c("y", "x1", "class", "school")] # slopes now on class
```

The predictor matrix shows which level it ended up on: of the columns carrying a -2, the leftmost is the one that holds the slopes. Future program versions will find a more convenient and less error-prone way to say which level gets the random slopes. Also, future program versions will allow random effects at all levels. The obstacle is not the estimation (`glmmTMB` fits slopes on two grouping levels without difficulty). The problem is that such a model cannot be written as a row of type codes at all: there is no place in it for a slope set per level. It would take a richer description than a predictor matrix holds, and therefore a method of its own. Note that random slopes on two levels at once is a demanding model and would require a sufficiently large sample at all levels!

Secondly, **the cluster bootstrap resamples the outermost level**. Drawing whole schools carries their classes along, which is what a nested design requires; where the grouping factors are crossed instead of nested, no such level exists and `draw = "boot"` stops and points at the Bayesian variant.

5.15.3 Variables that live at the cluster level

Some variables do not vary within a cluster at all: class size, ward type, the budget of a school. Imputing a variable like class size row by row would give one class different sizes. The `2lonly` methods exist for this case — `2lonly.pois` and `2lonly.nb` for counts, `2lonly.pmm` and `2lonly.norm` for the rest. They aggregate to one row per cluster, impute there, and hand the value back to every row of the cluster.

These are reached through the classical route only. The formula says which variables enter the model and at which level the random effects sit; it currently has no way of saying that the *target* lives at the cluster level. Writing `size ~ x1 + (1 | class)` therefore selects the ordinary two-level method, and `countimp` stops rather than let it through:

```

countimp: 'size' is constant within 'class', so it is a variable of that level,
but method "2l.poisson" imputes it row by row. The completed data would then
contain clusters whose 'size' differs between their own rows.
Use a cluster-level method instead, e.g. method = c(size = "2lonly.pmm")
(also 2lonly.norm, 2lonly.pois, 2lonly.nb), and code 'class' as -2 in that row
of the predictorMatrix.

```

The call it asks for names the method and codes the grouping variable as -2:

```

set.seed(20260927)
n_c <- 40; n_p <- 15
class <- rep(seq_len(n_c), each = n_p)
size <- rep(rpois(n_c, 22), each = n_p)      # constant within its class
x1 <- rnorm(n_c * n_p)
d2 <- data.frame(size, x1, class)
d2[size[d2$class %in% sample(n_c, 10)] <- NA] # ten classes missing entirely

pm2 <- matrix(0, ncol(d2), ncol(d2), dimnames = list(names(d2), names(d2)))
pm2["size", "x1"] <- 1
pm2["size", "class"] <- -2

imp_only <- countimp(d2, method = c(size = "2lonly.pois", x1 = "", class = ""),
                     predictorMatrix = pm2, m = 1, maxit = 1, printFlag = FALSE)

## every class carries one value, as it must
cd <- countimp_complete(imp_only, 1)
table(tapply(cd$size, cd$class, function(v) length(unique(v))))

```

1
40

With more than one grouping variable the outermost level the variable is constant within is the one used: a school-level variable is constant within its classes too, and imputing it per class would give classes of the same school different values.

5.16 Imputation of overdispersed two-level count data

Method `2l.nb` imputes incomplete data based on a two-level NB model (for details, see Kleinke & Reinecke, 2015b). The two-level NB model is estimated with `glmmTMB`, which replaced `glmmADMB` in version 2. The function requires the data in long format. Its `predictorMatrix` coding is the one every two-level method of this package uses. Allowed entries are listed in Table 6.

Table 6: Allowed entries in the `predictorMatrix`

numeric code	meaning
0	variable is not included in the imputation model
1	variable is included as a fixed effect
2	variable is included as a fixed <i>and</i> random effect
-2	class variable

The same model through the formula interface. The coding is the one of Table 5 or Table 6; only the family differs.

```
countimp(data, formulas = list(y ~ x1 + (1 + x2 | id)), family = nb())
```

`family = nb()` selects `2l.nb` because the formula carries a grouping term — the same formula without `(... | id)` would select the single-level `nb`.

5.16.1 Example

We use the model of Section 5.15 on the same data (`crim4l`, Section 5.3), with a negative binomial count part instead of a Poisson one. Only the family changes:

```

set.seed(20260907)
imp_2lnb <- countimp(crim4l,
                     formulas = list(DELINQ ~ FEMALE + RE + GY + TIME + (1 | ID)),
                     family = nb(),

```

```
m = 5, maxit = 5, printFlag = FALSE)
imp_2lnb$method[["DELINQ"]]
```

```
[1] "2l.nb"
```

One diagnostic is worth warning about here, because it looks obvious and is misleading. Comparing the variance-to-mean ratio of the imputed values against the observed ones does *not* tell you whether the count part needed the extra dispersion parameter:

```
vm <- function(x) var(x) / mean(x)
round(c(observed = vm(crim4l$DELINQ[!is.na(crim4l$DELINQ)]),
      `2l.poisson` = vm(unlist(imp_2lp$imp$DELINQ)),
      `2l.nb` = vm(unlist(imp_2lnb$imp$DELINQ))), 2)
```

observed	2l.poisson	2l.nb
3.57	3.77	4.80

The Poisson imputations are *already* overdispersed at the margin, because the random intercept adds variance on top of the conditional Poisson variance, and the negative binomial adds more still. A marginal ratio therefore cannot separate the two sources. Which model the count part needs is a question for the fit, not for the imputed values.

5.16.2 Simulation evidence

Kleinke & Reinecke (2015b) evaluated `2l.nb` on overdispersed two-level data, and the re-run in Kleinke & Reinecke (2026) repeats it against the current implementation. With the model assumptions met, the average estimates sit close to the generating values and coverage stays near the nominal level for every parameter, including the variance components.

5.17 Imputation of zero-inflated two-level count data

Methods `"2l.hp"` and `"2l.hnb"` impute zero-inflated two-level count data based on a two-level hurdle model. Analogously, methods `"2l.zip"` and `"2l.zinb"` create imputations based on a two-level zero-inflated Poisson or NB model respectively. The intercept is always treated as a random effect by default. The `.noint` variants keep the intercept fixed and take it out of the random part. Because there are two model parts (the zero and the count model respectively) there are three variants of each imputation function: `.noint.count`, `.noint.zero` and `.noint.both`. Each of the four families carries all three, with a bootstrap counterpart for every one, so `2l.zip`, `2l.zinb`, `2l.hp` and `2l.hnb` come to eight method names apiece (`help("mice.impute.2l.hnb")` lists them). On the formula route, each part states its own random-effects term, so the variant follows automatically from the two formulas specified by the user. The example below uses `hurdle_nb()`. `zi_poisson()`, `zi_nb()` and `hurdle_poisson()` work identically:

count part	zero part	method selected
(1 + TIME ID)	(1 + GY ID)	2l.hnb
(0 + TIME ID)	(1 + GY ID)	2l.hnb.noint.count
(1 + TIME ID)	(0 + GY ID)	2l.hnb.noint.zero
(0 + TIME ID)	(0 + GY ID)	2l.hnb.noint.both

`-1` in place of `0` does the same, as in `lme4`. It removes the intercept. What does *not* work is leaving the random-effects term out of the `zero` formula altogether: both parts have to name the same grouping level, and a call that omits it stops with a message.

Different sets of predictors can be specified for the zero and count parts of the model. Allowed entries in the `predictorMatrix` are listed in Table 7:

Table 7: Allowed entries in the `predictorMatrix`

numeric code	meaning
0	variable is not included in the imputation model
1	variable is included as a fixed effect

numeric code	meaning
2	variable is included as a fixed <i>and</i> random effect
3	variable is included as a fixed effect (count model only)
4	variable is included as a fixed and random effect (count model only)
5	variable is included as a fixed effect (zero model only)
6	variable is included as a fixed and random effect (zero model only)
-2	class variable

The same model through the formula interface. Two model parts need two formulas: `formulas` carries the count part, `zero` the zero part. Each takes its own random-effects term, so the two parts can differ in both their fixed and their random structure — something the numeric codes can express only through the eight-way table above.

```
## count part: x1 fixed and random; zero part: z1 fixed, random intercept
countimp(data, formulas = list(y ~ (1 + x1 | id)),
  zero = ~ z1 + (1 | id), family = hurdle_nb())
```

This produces the `predictorMatrix` entries `x1 = 4`, `z1 = 5`, `id = -2` — identical to coding them by hand.

5.17.1 Example

In this example we specify a model with a random intercept over `ID` in both model parts- The zero part takes its own formula through the `zero` argument. `FEMALE` and `GY` are the predictors in the zero model, `FEMALE`, `RE`, `GY`, and `TIME` are the count model predictors. The selection of predictors was not theory-driven. It is merely to illustrate that the zero and count model can have different predictors. The data set is `crim41` (Section 5.3):

```
data(crim41)
set.seed(20260907)
imp_2lhnrb <- countimp(crim41,
  formulas = list(DELINQ ~ FEMALE + RE + GY + TIME + (1 | ID)),
  zero      = ~ FEMALE + GY + (1 | ID),
  family    = hurdle_nb(),
  m = 5, maxit = 5, printFlag = FALSE)
imp_2lhnrb$method[["DELINQ"]]
```

```
[1] "2l.hnb"
```

```
c(observed = mean(crim41$DELINQ == 0, na.rm = TRUE),
  imputed   = mean(unlist(imp_2lhnrb$imp$DELINQ) == 0))
```

```
observed  imputed
0.6070743 0.5284861
```

Replace `hurdle_nb()` by `zi_nb()` for the zero-inflated counterpart, and by `hurdle_poisson()` or `zi_poisson()` for the Poisson versions. Writing the zero part's random-effects term without an intercept selects a `.noint` variant, as the table above sets out.

5.17.2 Simulation evidence

Kleinke & Reinecke (2015b) evaluated the two-level zero-inflation and hurdle methods and found them to recover the fixed effects and the variance components of the generating model when its assumptions hold. The re-run described in Kleinke & Reinecke (2026) repeats that design with the current implementation and reports the tables; it is the citable source for both.

5.18 Choosing a model and checking the imputations

Two questions arise around every imputation: which family fits the variable before anything is imputed, and whether what the functions produce is usable afterwards.

5.18.1 Choosing a family: `countimp_fit_diag()`

The family sections above each describe one model. `countimp_fit_diag()` weighs the candidates against an observed variable:

```
library(countimp)
data(crim4w)

countimp_fit_diag(BCRIM ~ FEMALE + RE + GY + HA, data = crim4w)
```

```
countimp: which count model fits the observed data?
1667 complete cases; mean 1.062, variance 3.964
dispersion:      3.24 -- overdispersed (given x; 3.73 marginal)
zeros:           60.6% observed vs 39.3% under Poisson -- excess zeros
```

	family	AIC	dAIC	P(Y=0)	status
	zero-inflated NB	4436.8	0.0	0.606	ok
	hurdle NB	4437.0	0.2	0.606	ok
	negative binomial	4439.2	2.4	0.603	ok
	Conway-Maxwell-Poisson	4559.9	123.1	0.516	ok
	zero-inflated Poisson	4733.3	296.6	0.607	ok
	hurdle Poisson	4733.4	296.6	0.606	ok
	Poisson	5670.4	1233.6	0.393	ok
	quasi-Poisson	--	--	0.393	ok

quasi-Poisson has no likelihood, hence no AIC; its dispersion estimate is still informative and it is excluded from the ranking.

recommended method: "zinb"

within 2 AIC units: hurdle NB

These fit the data about equally well. Hurdle and zero-inflated models differ in whether the zeros have one source or two -- a substantive question the data cannot settle. Choose on subject matter grounds, not on AIC.

It reports the variance-to-mean ratio, the observed share of zeros against the share a Poisson model would imply, and an AIC ranking over the families that converged. Quasi-Poisson has no likelihood and therefore no AIC; it is excluded from the ranking but its dispersion estimate is still printed.

The `status` column says how each candidate's fit ended. `ok` means it converged and is part of the ranking; anything else names the reason it is not — `did not converge`, `data contain zeros`, or a suggested package that is missing — and those candidates are repeated under a `not ranked` heading beneath the table. Every candidate converged here, so that heading does not appear. A candidate that does not converge is not a badly fitting one: usually the family is richer than the data support, as when a zero-inflated negative binomial is fitted to data with no overdispersion and θ is left unidentified.

The recommendation is printed in the first row. Where a hurdle and a zero-inflated model land within a couple of AIC units of each other (the `dAIC` column gives the difference between the two AIC values) the data cannot separate them, because the two differ in whether the zeros have one source or two. That is a question about the subject matter, not about fit (Section 5.10).

5.18.2 What happened during imputation: `countimp_diagnostics()`

Some things a method does on the way to a draw raise no error and show up in no convergence count. Those are the events worth recording, and they may be found in the diagnostics log. `countimp_diagnostics()` also flags draws the data do not support — a draw more than ten times the largest observed value — and *keeps* them, because discarding large draws until a small one arrives pulls the imputations downwards.

The checks themselves are made by `countimp_check()`, which runs once per draw and files its results in the log.

Two kinds of entry end up there. The first are the **automatic checks**, which `countimp_check()` runs on every draw (Table 8). Each group can raise several findings; a draw comes back as `ok` when none did, as `flagged` when something is worth knowing, and as `hard` when the draw cannot be trusted at all — a failed fit or a non-finite rate, coefficient or draw.

Table 8: What `countimp_check()` examines on every draw

What is examined	What is looked for
the fit	it failed, did not converge, has non-finite or extreme coefficients or standard errors, or a degenerate dispersion parameter
the rate μ	it is unavailable, non-finite, extrapolated beyond the observed predictors, or out of proportion to the observed data
the draws	there are none, they are non-finite, out of proportion, or far above the largest observed value

The second are the **substitutions** a method made on the way to a draw (Table 9). None of them is an error, none appears in a convergence count, and that is exactly why they are recorded. The ones listed here come from the companion methods, which impute the variables that are not counts; the count families record their own decisions in the same log, such as a dispersion parameter held at its floor.

Table 9: Substitutions recorded in the diagnostics log

What is recorded	Where it comes from
a ridge penalty where the predictors are collinear	<code>countimp.impute.norm()</code> , <code>.norm.nob()</code> , <code>.norm.predict()</code> , <code>.pmm()</code>
an imputation from the maximum-likelihood estimate where the logistic fit is singular	<code>countimp.impute.logreg()</code>
the same where the multinomial Hessian is singular	<code>countimp.impute.polyreg()</code>
the same where the proportional-odds Hessian is singular	<code>countimp.impute.polr()</code>
a proportional-odds model that could not be fitted and was replaced by a multinomial one	<code>countimp.impute.polr()</code>

Recording is on by default. What is worth doing before a run is clearing the log, because it survives across calls and would otherwise mix the entries of several imputations. `enable = FALSE` turns recording off, for a long run where it is not wanted:

```
countimp_diagnostics(reset = TRUE)

set.seed(1)
imp_diag <- countimp(crim4w,
  formulas = list(ACRIM ~ FEMALE + RE + GY + HA,
    BCRIM ~ FEMALE + RE + GY + HA,
    CCRIM ~ FEMALE + RE + GY + HA,
    DCRIM ~ FEMALE + RE + GY + HA),
  family = nb(), m = 2, maxit = 2, printFlag = FALSE)

countimp_diagnostics()
```

countimp imputation diagnostics: 16 draws recorded, 0 flagged
all draws passed the automated checks.

5.18.3 Observed against imputed: `compare.obs.imp()` and `compare.percent.count()`

The first places the summary statistics of the observed and the imputed part side by side, one entry per incomplete variable:

```
compare.obs.imp(crim4w, imp_diag)["BCRIM"]
```

```
$BCRIM
```

	Group.1	x.Min.	x.1st Qu.	x.Median	x.Mean	x.3rd Qu.	x.Max.
1	imputed	0.000000	0.000000	0.000000	1.366499	2.000000	30.000000
2	observed	0.000000	0.000000	0.000000	1.061788	1.000000	15.000000

The second does the same for the frequency of each individual count, as percentages. Its `counts` argument takes the *names of the variables* to tabulate, not the count values:

```
compare.percent.count(crim4w, imp_diag, counts = "BCRIM")
```

```
$BCRIM
```

		0	1	2	3	4	5	6	7	8	9	10	11	12
imputed		58.44	15.37	9.32	4.66	3.65	2.90	1.51	0.63	0.63	0.50	0.38	0.25	0.25
observed		60.65	17.28	7.32	4.68	3.36	2.58	1.26	0.96	0.60	0.36	0.24	0.06	0.36
		13	14	15	18	19	24	30						
imputed		0.13	0.25	0.38	0.25	0.25	0.13	0.13						
observed		0.12	0.06	0.12	0.00	0.00	0.00	0.00						

Neither is a test, and agreement is not the goal. Wherever the mechanism is not MCAR the incomplete cases differ systematically from the complete ones, so under MAR a gap between the two rows is *expected* (Section 5.9). What these tables are good for is the size and the direction of that gap.

Here the two functions say something the log does not. The log reports no flagged draw, and correctly so: its threshold is ten times the largest observed value, which for BCRIM would be 150. Yet the imputations run to 30 where the observed values stop at 15, and the percentage table shows imputed mass at counts that were never observed at all. The automated check and the reader's eye answer different questions, which is why the package offers both.

Contact

PD Dr. Kristian Kleinke
University of Siegen
Department of Psychology
Obergraben 23
D-57072 Siegen
+49 271 740 2974
mail@kleinke.net

References

- Audigier, V., & Resche-Rigon, M. (2018). *Micemd: Multiple imputation by chained equations with multilevel data*. <https://CRAN.R-project.org/package=micemd>
- Barnard, J., & Rubin, D. B. (1999). Small-sample degrees of freedom with multiple imputation. *Biometrika*, 86(4), 948–955.
- Bartlett, J. (2025). *Smcfcs: Multiple imputation of covariates by substantive model compatible fully conditional specification*. <https://CRAN.R-project.org/package=smcfcs>
- Boers, K., Reinecke, J., Seddig, D., & Mariotti, L. (2010). Explaining the development of adolescent violent delinquency. *European Journal of Criminology*, 7(6), 499–520. <https://doi.org/10.1177/1477370810376572>
- Burkoff, N., Metcalfe, P., Bartlett, J., & Ruau, D. (2024). *dejaVu: Multiple imputation for recurrent events*. <https://CRAN.R-project.org/package=dejaVu>
- Erlor, N. S., Rizopoulos, D., & Lesaffre, E. M. E. H. (2021). JointAI: Joint analysis and imputation of incomplete data in R. *Journal of Statistical Software*, 100(20), 1–56. <https://doi.org/10.18637/jss.v100.i20>
- Hilbe, J. M. (2011). *Negative binomial regression* (2nd ed.). Cambridge University Press. <https://doi.org/10.1017/cbo9780511973420>

- Keller, B. T., & Enders, C. K. (2026). *Blimp user's guide*. <https://www.appliedmissingdata.com/blimp>
- Kleinke, K. (2017). Multiple imputation under violated distributional assumptions – a systematic evaluation of the assumed robustness of predictive mean matching. *Journal of Educational and Behavioral Statistics*, 42(4), 371–404. <https://doi.org/10.3102/1076998616687084>
- Kleinke, K., & Reinecke, J. (2013). Multiple imputation of incomplete zero-inflated count data. *Statistica Neerlandica*, 67(3), 311–336. <https://doi.org/10.1111/stan.12009>
- Kleinke, K., & Reinecke, J. (2015a). Multiple imputation of multilevel count data. In U. Engel, B. Jann, P. Lynn, A. Scherpenzeel, & P. Sturgis (Eds.), *Improving Survey Methods: Lessons from Recent Research* (pp. 381–396). Routledge, Taylor & Francis. <https://doi.org/10.4324/9781315756288-46>
- Kleinke, K., & Reinecke, J. (2015b). Multiple imputation of overdispersed multilevel count data. In U. Engel (Ed.), *Survey Measurements. Techniques, Data Quality and Sources of Error* (pp. 209–226). Campus/The University of Chicago Press.
- Kleinke, K., & Reinecke, J. (2022). How to and how not to impute incomplete count data. In A. Hernández & I. Tomás (Eds.), *Proceedings from the 9th european congress of methodology* (pp. 86–92). Universitat de València. <https://doi.org/10.7203/PUV-OA-438-5>
- Kleinke, K., & Reinecke, J. (2026). *countimp: Multiple imputation of incomplete count data in R* [Manuscript in preparation].
- Kleinke, K., Reinecke, J., Salfrán, D., & Spiess, M. (2020). *Applied multiple imputation: Advantages, pitfalls, new developments and applications in R*. Springer International Publishing. <https://doi.org/10.1007/978-3-030-38164-6>
- Lambert, D. (1992). Zero-inflated poisson regression. With an application to defects in manufacturing. *Technometrics*, 34(1), 1–14. <https://doi.org/10.2307/1269547>
- Lee, J. A. (2025). *Accelmissing: Missing value imputation for accelerometer data*. <https://CRAN.R-project.org/package=accelmissing>
- Little, R. J. A., & Rubin, D. B. (2002). *Statistical analysis with missing data* (2nd). Wiley.
- Mullahy, J. (1986). Specification and testing of some modified count data models. *Journal of Econometrics*, 33(3), 341–365. [https://doi.org/10.1016/0304-4076\(86\)90002-3](https://doi.org/10.1016/0304-4076(86)90002-3)
- Muthén & Muthén. (2020). *Version 8.5 Mplus language addendum*. Muthén & Muthén. <https://www.statmodel.com/download/Version8.5LanguageAddendum.pdf>
- Muthén, L. K., & Muthén, B. O. (2017). *Mplus user's guide* (8th). Muthén & Muthén.
- 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.
- Reinecke, J., & Weins, C. (2013). The development of delinquency during adolescence: A comparison of missing data techniques. *Quality & Quantity*, 47(6), 3319–3334. <https://doi.org/10.1007/s11135-012-9721-4>
- Rubin, D. B. (1976). Inference and missing data. *Biometrika*, 63(3), 581–592. <https://doi.org/10.1093/biomet/63.3.581>
- Rubin, D. B. (1987). *Multiple imputation for nonresponse in surveys*. Wiley. <https://doi.org/10.1002/9780470316696>
- Salfran, D., & Spiess, M. (2018). Generalized additive model multiple imputation by chained equations with package *ImputeRobust*. *The R Journal*, 10(1), 61–72. <https://doi.org/10.32614/RJ-2018-014>
- Schafer, J. L. (1997). *Analysis of incomplete multivariate data*. Chapman & Hall. <https://doi.org/10.1201/97814139821862>
- Shmueli, G., Minka, T. P., Kadane, J. B., Borle, S., & Boatwright, P. (2005). A useful distribution for fitting discrete data: Revival of the Conway–Maxwell–Poisson distribution. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 54(1), 127–142. <https://doi.org/10.1111/j.1467-9876.2005.00474.x>
- StataCorp. (2025). *Stata multiple-imputation reference manual: Release 19*. StataCorp LLC; Stata Press.
- Su, Y.-S., Gelman, A., Hill, J., & Yajima, M. (2011). Multiple imputation with diagnostics (mi) in r: Opening windows into the black box. *Journal of Statistical Software*, 45(2), 1–31. <https://doi.org/10.18637/jss.v045.i02>
- van Buuren, S. (2012). *Flexible imputation of missing data*. Chapman & Hall / CRC. <https://doi.org/10.1201/b11826>
- van Buuren, S. (2018). *Flexible imputation of missing data* (2nd ed.). Chapman & Hall / CRC. <https://doi.org/10.1201/9780429492259>
- van Buuren, S., & Groothuis-Oudshoorn, K. (2011). MICE: Multivariate imputation by chained equations in R. *Journal of Statistical Software*, 45(3), 1–67.
- von Hippel, P. T. (2013). Should a normal imputation model be modified to impute skewed variables? *Sociological Methods & Research*, 42(1), 105–138. <https://doi.org/10.1177/0049124112464866>

- Wu, W., Jia, F., & Enders, C. (2015). A comparison of imputation strategies for ordinal missing data on likert scale variables. *Multivariate Behavioral Research*, 50(5), 484–503. <https://doi.org/10.1080/00273171.2015.1022644>
- Yu, L. M., Burton, A., & Rivero-Arias, O. (2007). Evaluation of software for multiple imputation of semi-continuous data. *Statistical Methods in Medical Research*, 16(3), 243–258. <https://doi.org/10.1177/0962280206074464>
- Zeileis, A., Kleiber, C., & Jackman, S. (2008). Regression models for count data in R. *Journal of Statistical Software*, 27(8), 1–25. <https://doi.org/10.18637/jss.v027.i08>