
# *PcAux*

***Kyle M. Lang, Jacob Curtis, Daniel E Bontempo***
**Institute for Measurement, Methodology, Analysis & Policy at Texas Tech University**
***May 5, 2017***

**What is PcAux? PcAux** is an R package that uses the Principal Component Auxiliary Variable method—developed by Howard, Rhemtulla, and Little (2015) — to extract auxiliary variables (PCAUX) from large datasets to use in missing data treatment.

**Why is it needed?** We need a modern, principled missing data treatment (i.e., MI or FIML) to adequately address the issues caused by nonresponse (Little & Rubin, 2002). These methods only perform optimally when all important predictors of the missingness are included as auxiliary variables (Collins, Schafer, & Kam, 2002). Choosing the correct set of auxiliary variables can be challenging in the "real-world." Too few auxiliaries can leave nonresponse bias uncorrected (Collins et al., 2002). Too many auxiliaries can induce needless noise into the imputed values or FIML-based parameter estimates (Graham, 2012). **PcAux** extracts auxiliary principal component scores, which succinctly summarize the important information contained in the incomplete data.

**Functions:**

Main functions:

**prepData** Initial data checks to clean the raw data. The initial data checks catch many common mistakes and help ensure the validity of the results.

**createPcAux** Extracts the principal component auxiliary variables from the incomplete data prepared by `prepData`.

**miWithPcAux** Creates multiple imputations using the principal component auxiliary variables created by `createPcAux`.

Depending on your application you may also find these functions and components useful:

**mergePcAux** Merge PcAux variables with the raw data frame. Using a PcAuxData object returned by createPcAux, you can append PcAux variables to the original dataset. This is useful if you wish to use other software to do FIML analyses using the PcAux. For example, PcAux variables can be declared as AUXILIARY variables in MPlus, or used as saturated covariates in other packages.

**getImpData** Extract multiply imputed datasets from a PcAuxData object returned by miWithPcAux. In addition to saving the object for archival purposes, you will likely want to write code that extracts the imputed datasets for analysis, or storage in external file(s).

**makePredMatrix** Make a predictor matrix. In some cases you may want to call mice() directly instead of using the PcAux::miWithPCAux() function. This function returns the PcAux variables for use as a predictor matrix in mice(). See also mergePcAux.

**inspect** Access fields of a PcAuxData object. This can be useful if troubleshooting is needed, or to extract information for reports to an end-user. For example, lists of collinear or constant variables.

**iris2** Dataframe with a modified version of the Fisher/Anderson iris data. This is included in the PcAux library so you can test PcAux functions. This data is also used in vignettes in the PcAux help pages.

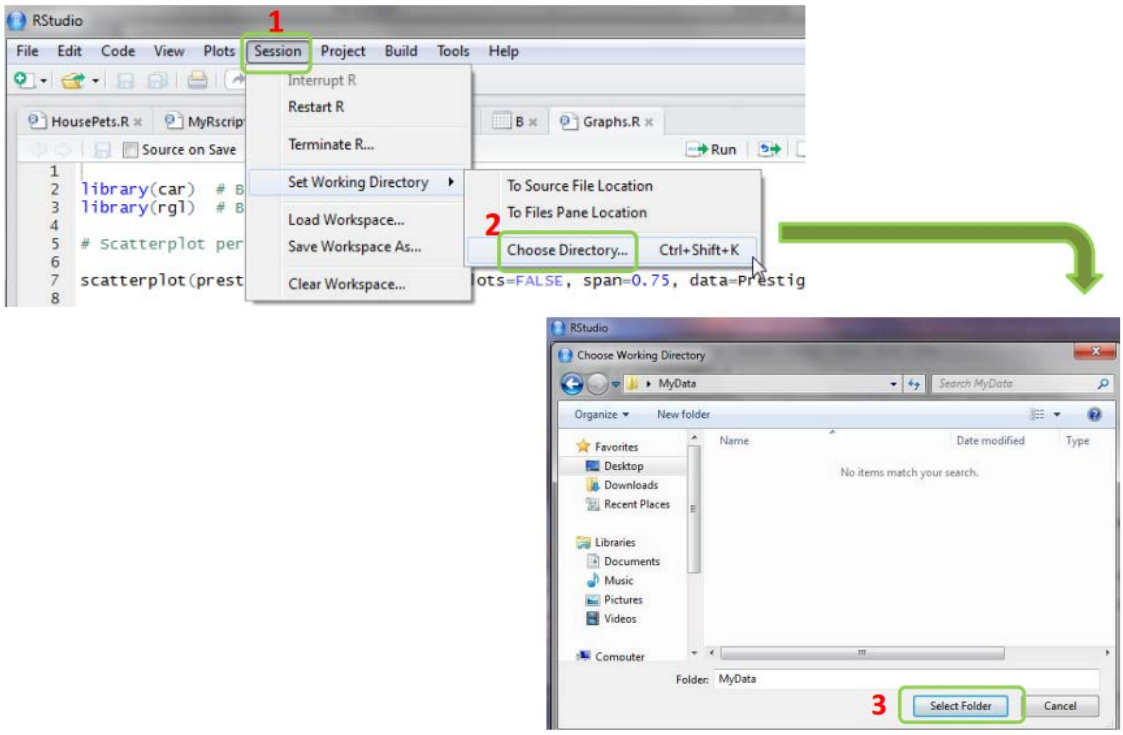
**Before you begin: Install and load the PcAux package into R, and read in your data**

To ensure that you're working with the latest version of **PcAux**, we recommend using the `install_github()` function from **devtools** to install it directly from github. Executing the code provided in Appendix A (i.e., by copying it into the R console) should automatically do all the heavy lifting for you.

Next, you need to load the **PcAux** library for use in the current R session. Installation is done once, but "loading" is needed each session.

```
library(PcAux)
```

Now that **PcAux** is installed and loaded, you should read in your data. At this point, you may want to change your working directory to the folder that contains your data so you don't have to explicitly provide R with the full file path to your data. Assuming you're using RStudio, you can change the working directory with drop-down menus by: clicking "Session" -> "Set Working Directory" -> "Choose Directory" and navigating to the folder wherein your data live. Finally, click "Select Folder" to change the location of your working directory to the selected folder. See pictures below:




Next, read your data in:

```
myData <- read.csv("myData.csv")
```

Now that you have **PcAux** installed and loaded and have read in your data, you are ready to start.

**Step 1: Run the `prepData` Function**

The `prepData` function returns a *PcAuxData* object with a copy of the data, all arguments passed to `prepData`, and fields summarizing data preparation and checks necessary for extracting principal component auxiliaries. The process entails some initial data checks, based on each variable's type. So, `prepData` requires information about the variable types of your data.

- **Nominal**: Categorical variables without meaningful ordering
  - Analyzed as unordered factors
  - Must treat nominals (e.g., No/Yes) as categorical (failure to do so will produce badly biased results)

```
myNoms <- c("nom1", "nom2")
```

- **Ordinal**: Categorical variables with meaningful ordering
  - Likert Scales, counts
  - Analyzed as ordered factors

- Sometimes, ordinals can be treated as continuous when they have many levels (especially counts, for example more than 8 levels). See Appendix C for code that can assist in preparing count variables.

```
myOrds <- c("ord1", "ord2", "count1" )
```

Any variable that is not flagged as nominal, ordinal, or ID in your dataset is assumed to be continuous.

In addition to defining the measurement level of discrete variables, there are several special classifications of variable we need to define:

- **ID:** Identification variables
  - Excluded from the analysis and returned in the final imputed datasets unaltered

```
myIds <- c("id1", "id2")
```

- **Grouping**: Categorical or Continuous variables that define groups
  - Used to create strata for group-mean substitution, if necessary
  - Continuous variables are binned before creating the strata

```
myGroups <- c("nom1", "nom2", "ord1")
```

- **Nuisance**: Variables that should not enter the missing data analysis
  - For example, open-ended character responses
  - Excluded from the analysis and returned in the final imputed datasets unaltered

```
myTrash <- c("y1", "y2", "z2")
```

- **Moderators:** Important moderating variables.
  - Interactions that you want to include in the inferential models must be represented in the imputation process
  - The variables defined as “moderators” will be used to define interaction terms that get incorporated into the PCAUX scores.
  - See note under Step 2 for details on how these variables are included in the analysis.

```
myMods <- c("nom1", "ord2", "w3")
```

*Note:* You need to change the variables in the examples above to the variables in your dataset. To see a list of all your variables, you can use the `colnames` function:

```
colnames(myData)
```

With these lists of variable types, you are ready to run the `prepData` function.

If you have scripted operations to read data and prepare arguments for `prepData`, you may want to execute the call to `prepData` by itself. There are special considerations when an interactive step is embedded in a script. Generally, these scripts can be sourced, but not submitted or pasted into the console. Behavior also varies depending on which development environment (e.g., **Emacs**, **NotePad++**, **RStudio**) you might be using.

The `prepData` step can take a long time (e.g., several hours) for large datasets with many categorical variables, please be patient.

```
pcAuxDat <- prepData(rawData    = myData,
                     moderators = myMods,
                     nomVars    = myNoms,
                     ordVars    = myOrds,
                     idVars     = myIds,
                     dropVars   = myTrash,
                     groupVars  = myGroups)
```

Among other things, the `prepData` function is checking for constant variables, highly collinear variables, variables with potentially problematic number of levels, and variables with high missing data rates. When `prepData` executes these data checks, several messages will print to the screen.

With no issues, the output will look like this:

```
Checking inputs' validity...
Complete.

Checking data and information provided...
--Examining data...done.
--Typing data...done.
--Casting data...done.
--Centering continuous data...done.
Complete.

Finding and addressing problematic data columns...
--Checking for high PM...done.
--Checking for empty columns...done.
--Checking for constant columns...done.
Complete.

Examining data for collinear relationships...
Complete.
```

Any issues will be flagged with R warnings. Some of these warnings will simply notify you of actions **PcAux** has taken automatically:

```
Warning: The following variables have no observed responses and have been
excluded from the data analysis: z1.
```

```
Warning: The following data columns are constants and have been excluded from
the data analysis: x3.

Warning: The following variable pairs are bivariate collinear: {y4, x2}, {y5,
x1}, so y4, y5 will be excluded from further analysis.
```

When a variable is "excluded from further analysis" this only means it will not be used to obtain PCAUX variables. **This does not mean the variable cannot be imputed.** Only empty variables cannot be imputed. If imputation is done with the **PcAux** package, constant variables will be completed with the constant value.

**Read each warning and make sure that the action PcAux has taken is appropriate and expected.**

- These are just warnings, not errors, so don't freak out.

Some warnings will explicitly ask you for input. For example:

```
Warning: The following variables have been declared as nominal: w2, but they
take: 500 levels, respectively.

Do you want to continue the analysis? (y/N)

Warning: The following variables have been declared as ordinal: w1, but they
take: 500 levels, respectively.

Do you want to continue the analysis? (y/N)

Warning: The following variables have been declared as continuous: nom2, but
they only take: 4 levels, respectively.

Do you want to continue the analysis? (y/N)
```

Type your response directly into the R console.

- y, yes, n, no, or any capitalization-based permutation are acceptable

Warnings convey important information about potential issues with the analysis…Don't Ignore the Warnings!

- Warnings about too many or too few levels for the declared variable types may indicate a mistake in your assignment of variable types.
- Warnings about constant, empty, or mostly missing columns may indicate coding errors.
- The analysis will often need to be restarted (repeatedly) after fixing the unexpected issues caught by the initial data checks.

Data and settings are passed between **PcAux** functions using an object of class *PcAuxData*. This object contains the cleaned version of *myData* and whatever metadata has accumulated from input or processing. Using the *PcAuxData* object in subsequent functions will change (augment) the object.

Therefore, it can be useful to back up the *PcAuxData* object to permit restarting at a previous step. You can do this using the copy member function:

```
frozenPcAuxData1 <- pcAuxData$copy()
```

**Step 2: Run the `createPcAux` Function**

With the cleaned data, we can create the principal component auxiliary variables by running the `createPcAux` function. Depending on several factors (e.g., the size of the dataset, the number of categorical variables, the number of linear PCAUX requested, etc.), this could take anywhere from minutes to weeks to run.

Before extracting components, a single imputation is needed. For large datasets, the predictor matrix for each variable can be very large. Also, because the predictors can themselves have missing data, several iterations are needed. This will take some time.

To include necessary interactions and non-linear data moments, the dataset is highly expanded before extracting component scores (exactly how this expansion is done depends on the values provided to the "interactType" and "maxPolyPow" arguments in `createPcAux`; see documentation of `createPcAux` for more details). For larger datasets this can mean considerable RAM and CPU demands. If you have problems, you should try moving to the largest machine available to complete this step. It can be efficient to complete the interactive `prepData` step on your desktop machine, and then run the `createPcAux` step on a server or cluster where it can run for a long time. These considerations are directly related to the size of your dataset. (There are also some options that minimize memory demands at the expense of precision; you should review the documentation for `createPcAux`'s "pcaMemLevel" argument.)

```
pcAuxDat <- createPcAux(PcAuxData    = pcAuxDat,
                        nComps       = c(0.5, 0.1),
                        interactType = 2)
```

NOTE: The "nComps" argument take a two-element vector defining the number of PCAUX scores to extract. The first element defines the number of "linear" PCAUX to extract while the second element defines the number of "nonlinear" PCAUX to extract. The second element is ignored when the "interactType" argument is set to 0 or 1. The entries in the "nComps" vector can be given in terms of counts of PCAUX to extract or in terms of the proportion of variance explained by the extracted number of PCAUX (as shown above). See the documentation of `createPcAux` for more details.

When all goes well, the output from `createPcAux` will look like:

```
Checking inputs' validity...
Complete.

Doing initial, single imputation...
--Constructing predictor matrix...done.
--Creating method vector...done.
```

```
--Filling missing values...done.
All variable successfully imputed in first pass.
Complete.

Calculating linear principal component scores...
Complete.

Computing interaction and polynomial terms...
Complete.

Calculating nonlinear principal component scores...
Complete.
```

After running the `createPcAux` function, the returned *PcAuxData* object contains the full set of principal component auxiliary variables.

If you want an augmented, incomplete dataset that you can analyze outside of **PcAux** (e.g., for use with FIML estimation), you can use the `mergePcAux` function:

```
newData <- mergePcAux(PcAuxData = pcAuxDat,
                      rawData   = myData)
```

NOTE: *newData* is not a *PcAuxData* object; it is a data frame containing the data provided to the "rawData" argument augmented with the PCAUX scores from the *PcAuxData* object provided to the "PcAuxData" argument.

**Step 3: Run the `miWithPcAux` Function**

The usual **PcAux** workflow concludes by running the `miWithPcAux` function to create multiple imputations using the PCAUX scores produced by `createPcAux`. Because the only variables used in the multiple imputation predictor matrix are the PCAUX variables, and because the PCAUX variables have no missing data, each imputation takes much less time than the single imputation performed by `createPcAux`.

```
pcAuxDat <- miWithPcAux(rawData   = myData,
                        PcAuxData = pcAuxDat)
```

NOTE: By default, the imputed datasets are returned as a list in which each element contains a unique imputed dataset. This format works best for analyses conducted in R. However, other programs like **SPSS** or **SAS** require a single stacked dataset. To get `miWithPcAux` to return that, you need to change the "compFormat" argument.

```
pcAuxDat <- miWithPcAux(rawData    = myData,
                        PcAuxData  = pcAuxDat,
                        compFormat = "long")
```

When all goes well, the output from `miWithPcAux` will look like this:

```
Checking inputs' validity...
Complete.

Checking data and information provided...
--Examining data...done.
--Typing data...done.
--Casting data...done.
Complete.

Finding and addressing problematic data columns...
--Checking for high PM...done.
--Checking for empty columns...done.
--Checking for constant columns...done.
Complete.

Multiply imputing missing data...
--Constructing predictor matrix...done.
--Creating method vector...done.
--Imputing missing values...

<< Printed Output From mice would go here >>

--done.
Complete.
```

The multiply imputed datasets live in the *miDatasets* field of the *PcAuxData object*. They can be accessed by running:

```
impData <- pcAuxData$miDatasets
```

As with all **PcAux** functions, any issues will be flagged with R warnings.

One warning that can occur in both `prepData` and `miWithPcAux` deserves special consideration.

- A warning like the following occurs when your data contain variables with very few observations:

```
Warning: The following variables have fewer than 25 observed responses: z2.
Would you like to remove them from the analysis? (Y/n)
```

When `prepData` returns this warning, you should remove the variables in question by answering "yes."

- Keeping these variables adds little information to the PCAUX scores
- Keeping these variables can induce spurious collinearity findings

When `miWithPcAux` returns this warning, answer "yes" only if you do not want to impute the missing data on the variables in question.

If you are doing your analysis in R, now that you have the accurately imputed datasets contained in `impData`, you are ready to do your analysis. If you are doing your analysis in other software, you will need to export it.

## References


Collins, L. M., Schafer, J. L., & Kam, C. M. (2001). A comparison of inclusive and restrictive strategies in modern missing data procedures. *Psychological methods*, *6*(4), 330-351.

Graham, J. W. (2012). *Missing data: Analysis and design.* New York: Springer.

Howard, W. J., Rhemtulla, M., & Little, T. D. (2015). Using principle components as auxiliary variables in missing data estimation. *Multivariate Behavioral Research, 50,* 285-299.

Little, R. J., & Rubin, D. B. (2002). *Statistical analysis with missing data.* Hoboken, NJ: Wiley.

**Appendix A: Code to install PcAux**

The following code will automatically make sure that you have the correct dependencies installed and then install the latest stable version of **PcAux** from Github using the `install_github` function from **devtools**. This is the recommended way of installing **PcAux**.

```
### Title:    Install PcAux from Github
### Author:   Kyle M. Lang
### Created:  2016-JUN-22
### Modified: 2017-MAY-04
### Notes:    This script will install the latest version of PcAux

## Check for dependencies:
depVec  <- c("devtools", "mice", "ICC", "vcd", "rlecuyer")
depTest <- suppressWarnings(
    unlist(lapply(depVec, require, character.only = TRUE))
)

## Install missing dependencies:
if(any(!depTest))
    install.packages(depVec[!depTest], repos = "http://cloud.r-project.org")

if(!depTest[1]) library("devtools")

## Use devtools::install_github to install the latest version of quark
install_github("PcAux-Package/PcAux/source/PcAux")
```

**Appendix B: Exporting Data**

Assuming impData contains a single data frame of stacked imputed datasets (i.e., because you specified *compFormat = "long"* in the `miWithPcAux` function), you can save a single stacked dataset for use in programs like **SPSS** or **SAS** using the `write.csv` function:

```
write.csv(impData, file = "impData.csv")
```

If impData contains a list of imputed datasets (i.e., the default) and you need to save separate data files to use in **Mplus**, execute the following steps:

1. Generating a list of variable names. You can copy these variable names into **Mplus**' "NAMES ARE" section.

    a. Use the `write.table` function to save the variable names from the first imputed dataset as a .txt file:

```
write.table(colnames(impData[[1]]),
            file      = "variable_names.txt",
            row.name  = FALSE,
            col.names = FALSE,
            quote     = FALSE)
```

2. Writing out each imputed dataset as a separate .csv files

    a. The code below will save the first dataset as "impData1.csv", the second one as "impData2.csv" and so on. You can change the names by modifying the value assigned to the *nameStem* variable. You can also change what NAs will be coded as by modifying the value assigned to the *naValue* variable.

    b. In addition to saving the datasets, the code below also saves a .txt file containing the dataset names. You will specify this file ("impFileList.txt") on the "FILE IS" line of your **Mplus** syntax.

```
nameStem <- "impData"
naValue  <- "-999"

for(m in 1 : length(impData)) {
    fileName <- paste0(nameStem, m, ".csv")

    write.table(impData[[m]],
                file      = fileName,
                row.names = FALSE,
                col.names = FALSE,
                sep       = ",",
                na        = naValue)
```

```
    write.table(fileName,
                file      = "impFileList.txt",
                row.names = FALSE,
                col.names = FALSE,
                append    = TRUE,
                quote     = FALSE)
}
```

**Appendix C: Preparing Count Variables**

If you know which count variables have just a few levels, you can simply add these names into the list of ordinal variables. For large datasets, however, it can be useful to have code to do this for you.

The following code will check a list of count variables for number of levels, and generate a list of count variable names that have less than 8 levels, and can reasonably be treated as ordinal (note that 8 is an arbitrary threshold that you may want to set higher or lower, depending on your problem).

```
countNames  <- c("count1", "count2", "count3")
countLevels <- apply(myData[ , countNames], 2,
                     FUN = function(x) length(unique(na.omit(x)))
                     )
ordCounts   <- names(countLevels)[countLevels <= 8]
```

Using just the non-missing levels of each variable, the number of unique patterns is obtained. The length function gives the count of unique patterns.

The apply function applies this code to each count variable identified by countNames, and returns a list of corresponding count levels. Each list element contains a count, and is named for the variable.

The logical indexing on the final line extracts just the variable names where there are 8 or fewer levels.