A Tutorial Introduction to PEP

Ben Klemens, Rolando Rodríguez

February 25, 2013

PEP is intended to unify several aspects of survey processing, key being editing, imputation of missing data, and disclosure avoidance; several other steps along the way are also supported.

This paper is a tutorial introduction, and if you have this tutorial, you should also have a working installation of PEP and a sample directory including `demo.spec`, `demo.R`, and `dc_pums_08.csv`. The `.csv` file is a base data set to be taken through full processing: over the course of this tutorial, you will read the text file into a database, check for errors, impute values for missing data, and prepare synthetic data ready to ship to the public.

1 System basics

We break the survey processing program into two parts: the specification of details—such as which file to read in, what values should be top-coded, or the full description of the model used for filling in missing data—and the actual prodecure to be run by the computer. The specification of details will go into a plain text file, and we expect you will find that segregating the specification to a separate file makes it easier to write for you and more readable for you and for any potential auditors in the future. The actual execution of procedures happens via R, an interactive and reasonably user-friendly system for doing the sort of statistics you will be doing.

The work flow for PEP is thus slightly different from the flow in systems that make no distinction between specification information and procedural code. You can expect to have two windows open as you work: one text editor with the spec file you are putting together, and R's window for executing a step and checking the results.

As for the choice of text editor, there are literally hundreds of options. If you don't have a favorite text editor, We recommend `kate`, but if you do, feel free to use it wherever we mention Kate.

A basic spec file In the directory where you found this document, you will find a file named `demo.spec`, which we suggest you open now with Kate. To facilitate discussion, here is a snippet:

```
database: test.db

input {
input_file :dc_pums_08.csv
output_table: dc_pums
}
```

First, we made the spaces erratic in this file to show you that spaces don't matter. But you should have no problem reading the file's specifications. The database is set to be `test.db`. The settings for the input segment of the process are surrounded by {curly braces}, and set the input file for the process to be the file provided with this tutorial, and the output to this segment to a table named `dc_pums`. You can see from looking over the spec file that it is almost entirely a series of one-line settings organized by groups, and sometimes subgroups.

The order of the file is also not important. It's common sense to put blocks in the order in which they appear in the pipeline of data analysis, with the `input` step first and the summarize-for-output steps last, but PEP won't enforce that. The only thing that is not negotiable: the first non-comment, non-blank line of the spec has to specify a database file. The system opens the database first, and writes to a table named `keys` as it reads your settings, which is why it first needs a database name before everything else. Only after everything is read in does any processing happen.

We'll get to all the things you can put in the spec file below, but for now, you probably want to see something actually run. For this, you will need R.

R When you start R (from the directory where the data is located), you are left at a simple command prompt, >, waiting for your input. PEP extends R via a library of functions for survey processing, but you will first need to load the library, with:

```
library(pep)
```

[You can cut crimson-bordered code blocks and paste them directly onto the R command line, while blue-bordered blocks are spec file samples and would be meaningless typed out at the R command prompt.]

Now you have all of the usual commands from R, plus those from PEP. You only need to load the library once per R session, but it's harmless if you run `library(pep)` several times.

Now load the specification file you began inspecting above:

```
read_spec("demo.spec")
```

All of the settings in the file are now known to the system, so you need not specify any new setting on the command line (though you can if something changed from the last time you ran `read_config`). So to read the CSV file to the database, all you have to type in R is

```
doInput()
```

and the system knows the names of the inputs and outputs.

The data is the ACS's 2008 PUMS file for DC (with a few glitches that we inserted so there would be something for later steps to clean).

Showing the data Now that you have the data in place, you probably want to have a look at it.

Data is stored in two places: the database, and R data frames. Database tables live on the hard drive and can easily be sent to colleagues or stored in backups. R data frames are kept in R's memory, and so are easy to manipulate and view, but are not to be considered permanent. Database tables can be as large as the disk drive can hold; R data frames are held in memory and can clog up memory if they are especially large.

You can use PEP's `show_db_table` function to pull a part of a database table into an R data frame. You probably don't want to see the whole table, so there are various options to limit what you get. Some examples:

```
teaTable("dc", cols="AGEP, PUMA", where="PUMA=104")
teaTable("dc", cols="AGEP, PUMA", limit=30, offset=100)
```

The first example pulls two columns, but only where `PUMA=104`. The second example pulls 30 rows, but with an offset of 100 down from the top of the table. You will probably be using the `limit` often; the `offset` allows you to check the middle of the table, if you suspect that the top of the table is not representative.

In fact, there are still more options. Rather than listing them all here, you can get them via R's help system:

```
?teaTable
```

This `?name` form should give you a help page for any function, including PEP functions like `?doRaking` or `?doMImpute`. [Yes, PEP function documentation is still a little hit-and-miss.] Depending on R's setup, this may start a paging program that lets you use the arrow keys and page up/page down keys to read what could be a long document. Quit the pager with `q`.

The `show_db_table` function creates an R data frame, and, because the examples above didn't do anything else with it, displays the frame to the screen and then throws it out. Alternatively, you can save the frame and give it a name. R does assignment via `<-`, so name the output with:

```
p104 <- teaTable("dc", cols="AGEP, PUMA", where="PUMA=104")
```

To display the data frame as is, just give its name at the command line.

```
p104
```

But you may want to restrict it further, and R gives you rather extensive control over which rows and columns you would like to see.

Another piece of R and spec file syntax: the `#` indicates a comment to the human reader, and R will ignore everything from a `#` to the end of the line.

```
p104[1,1]           #The upper-left element
p104[1,"AGEP"]      #The upper-left element, using the column name
p104[ ,"AGEP"]      #With no restriction on the rows, give all rows---the full AGEP vector
p104[17, ]          #All of row 17

minors <- p104[, "AGEP"] < 18
minors              #A true/false vector showing which rows are under 18.
p104[minors, ]      #You can use that list of true/falses to pick rows. This gives all rows
```

Declaring variables We return to the spec file. You need to declare to the system on which fields you will want consistency checks run, and what values they can take. This provides an additional check: if the field takes on a value that you did not declare, the system will let you know.

You can see the declarations at the top of the sample spec file. Numeric values can be given a range, but you can include a comma-separated list of ranges, numbers, or text:

```
|AGEP 0-116
|SEX 0-2, NA
```

You may also want to recode variables, such as for changing ages to categories. In the `recodes` section of the spec, you will find this recode:

```
recodes {
CATAGE {
1|AGEP between 0 and 15
2|AGEP between 16 and 64
3|
}
}
```

If `AGEP` is between zero and fifteen, then the `CATAGE` variable takes on a value of 1; 16 to 64 becomes 2; and everything else becomes 3. You can do consistency checks and other steps below on recodes just like any other variable.

2 Consistency

Consistency checks are simply assertions that some elements in a data set are not acceptable and probably an error, such as age < 10 while income > 100k. Consistency checks do not state anything about how the error is to be reconciled, but just check whether there is a problem.

They are checked repeatedly throughout the process. For example, if a value is imputed, then it is checked to make sure that the inserted value isn't inconsistent. The imputation step does a consistency check on every imputation to make sure that it is not generating unacceptable data. Raking, discussed below, uses the consistency checks as structural zeros.

The consistency checks go in a group named `check`. Here are a few examples from the sample file:

```
checks {
    SEX=0 and HISPF=0
    AGEP > 0
    AGEP > 90 => agep = 90
}
```

You can verify from the recode definitions in the spec file that the first condition (`sex=0 and hispf=0`) can never be true, so if this check is hit, something has truly gone wrong. The second probably won't appear in input data, but a model-based impuation could conceivably produce a negative imputed value, in which case this constraint would veto the value. The last line is a *pre-edit*: a condition that must never be true (`agep > 90`) plus a specific step to take should the condition hold for a record (set the age to 90).

The examples to this point should be enough to give you some image of the form of the checks. As for the exact grammer, the consistency rules are in a standard databse-oriented language named SQL (Structured Query Language), which means you can use your favorite search engine to find SQL tutorials, SQL references, et cetera. Anything that can be put in the `where` clause of a query can be a consistency check (because that's how the system prepares these checks internally). Those familiar with SQL will see that we've already pulled this trick: the recodes are also SQL clauses that used the SQL `between` keword to specify a range of variables.

An *edit* consists of two steps: (1) identifying something that fails a consistency check, and (2) filling in consistent data to replace the inconsistent. Except for pre-edits, PEP separates the two steps. `doCheck()` does step 1, and `doMImpute()` does step two. The separation allows you to choose from any of a variety of means of filling in questionable data points. Also, you will see that entries at risk of disclosure get the same treatment: they are simply flagged and then imputed later on.

We recommend saving the pre-edit form including a specific rule for replacement to cases where there is absolutely no ambiguity. The example here is a top-code, where all values over a given value are cut down to the limit; the action is unaffected by the value of other variables, and does not involve random draws (such as via hot-deck) or statistical modeling (such as drawing from a Normal distribution). Both forms will save the condition to a matrix of data constraints, and those data constraints will be re-checked after every change to the data, so there is no need for you the user to call for a re-check after imputations, swappings, and so on.

3 Flagging for disclosure

For a given crosstab, there may be cells with a small number of people, perhaps only one. This section flags those cells for later handling.

Any combination of variables could be a crosstab to be checked, but flagging typically focuses on only a few sensetive sets of variables. Here is the section of the spec file describing the flagging. The `key` list gives the variables that will be crossed together. With `combinations: 3`, every set of three variables in the list will be tried. The `frequency` variable indicates that cells with two or fewer observations will be marked.

We are calling this specific form of disclosure avoidance *fingerprinting*, so after this segment of the spec file is in place, call *doFingerprinting()* from R to run the procedure. The output is currently in a database table named *vflags*.

```
fingerprint {
key{
CATAGE
SEX
PUMA
HISPF
ESR
PWGTP
}
frequency: 2
combinations: 3
}
```

4 Imputation

Imputing a missing or questionable data point first requires making a statement about the model by which the data is drawn. The Normal distribution is always a popular claim: we could say that age has a Normal distribution, then estimate the parameters of the Normal (μ and σ) from all people of the same race, gender, and geographic region. Once we have fit this distribution with this data, we can draw a value for the missing age from the distribution.

Or, the Randomized hot deck asserts that estimating a Normal distribution is unnecessary, and that the best model to describe a missing age is simply the list of other ages in the race/gender/geography category; we thus fill in the missing age by randomly drawing an age from the list of other ages in the category. Or, the linear regression models claim that one variable is a function of others.

The process breaks down to a few standard steps in all cases:

- Specify a category: what set of ages, geographies, et cetera should we use as the subset for specifying the model? The sample spec file uses age $\times$ sex.
- Specify a model: As above, you may want to use a distribution, a hot-deck imputation (which is equivalent to the multinomial distribution), or a linear regression. See below for more on this step.

- Draw an imputation: once the model is estimated, PEP will make a draw from the model to produce an imputation. In fact, it will produce `draw_count` draws. These multiple imputations are stored in the database, and you have the choice of using the mean, only the first draw, or of using the imputations to report a confidence band for any given imputed value.

Each variable will need its own model specification—it is unlikely that income and age were truly generated from the same process. The spec file includes a Hot deck and Ordinary least squares model. The Hot deck model simply requires specifying that that is your choice of model (given that the subcategories are already specified). The OLS model requires specifying the explanatory variables for the regression. This is another SQL query, and any `select` syntax is OK, as are typical math functions. The example here simply regresses age on one variable, but with the right variables and temperament, you could specify a regression like `vars: log(income), sex`.

```
models{
    sex { method: hot deck
    }

    agep { method: ols
              vars: sex
    }
}
```

Here is the current list of options for imputation methods: `normal`, `multivariate normal`, `lognormal`, `hot deck` (aka `multinomial`), `poisson`, `ols`, `logit`, `probit`, `kernel_density` estimate.

The results are in a table named `filled`. At the moment, the consistency checks are disabled, so you may actually see negative ages.

5 Raking

Raking is a method of producing a consistent table of individual cells beginning with just the column and row totals. For this tutorial, we will use it as a disclosure-avoidance technique for crosstabs. The column sums and row sums are guaranteed to not change; all individual cells are recalculated. Thus, provided the column totals have passed inspection for avoiding disclosure, the full table passes.

The key inputs are the set of fields that are going to appear in the final crosstab, and a list of sets of fields whose column totals (and cross-totals) must not change.

In the spec file, you will see a specification for a three-way crosstab between the `PUMA`, `rac1p`, and `catage` variables. All pairwise crosstabs must not change, but any other details are free to be changed for the raking.

```
raking {
all_vars: puma|catage | rac1p

contrasts{
   catage | rac1p
 puma | rac1p
 puma | catage
}
}
```

As you have seen a few times to this point, once you have the spec file in place you can call the procedure with one R function, which in this case is `doRaking()`. But there are several ways to change the settings as you go, so that you can experiment and adjust.

The first is to simply change the spec file and re-run. To do this, you will have to call `read_config` again:

```
read_spec("demo.spec"); doRaking()    #Run two commands on a line by ending the first with a
read_spec("demo.spec"); doRaking()    #Hit the up-arrow to call up the previous line.
```

Everything you can put in a spec file you can put on the command line. The help system will give you the list of inputs (which will also help with writing a spec file), and you can use those to tweak settings on the command line:

```
?doRaking                     #What settings are available?
doRaking(max_iterations=2)  #What happens when the algorithm doesn't have time to
converge?
```

6 Putting it all together

Along with `demo.spec`, you will also find `demo.R`, which is an R script, simply consisting of all the steps above. This is your permanent record of the process, which you can modify, re-run later, or hand off to colleagues or an auditor. A year from now, when you are trying to remember what you did to the survey, the spec file plus the R script will give you a complete, precise answer.

You can run an R script in two ways. From the R command line, this is referred to as *sourcing*:

```
source("demo.R")
```

If you work from a shell prompt, you can call the script without entering R:

```
shell>> R CMD BATCH demo.R
```

This runs silently, and produces a file named `demo.Rout`, which you can inspect to make sure everything ran smoothly.