

Examining the R `datastep()` Function

David J. Bosak, Archytas Clinical Solutions

ABSTRACT

Did you know there is a `datastep()` function for R? It is found in the "libr" package. This function replicates many of the basic features of the SAS DATA STEP. It processes data row by row, and allows you to execute an arbitrary code block on each row. Variables can be created on the fly, just like a SAS DATA STEP. You can also drop, keep, and rename variables, retain variables, and create variable arrays. It supports set, merge, and by operations. It also supports `output()` and `delete()` functions to control which rows are sent to the output dataset. The purpose of the paper is to examine the capabilities of this useful function.

INTRODUCTION

For a SAS® programmer, the SAS DATA STEP is the fundamental tool for manipulating data. It is totally ingrained in the way SAS programmers think. Therefore, it is not surprising that the first thing many SAS programmers ask when they come to R is: Is there an R equivalent of a SAS DATA STEP? Fortunately, there is!

The `datastep()` function from the [libr](#) package aims to give R programmers the basic capabilities of a SAS DATA STEP. The `datastep()` function allows you to process data row-by-row, and execute an arbitrary set of programming statements on each row. This function makes it much easier for SAS programmers to get things done in R. It is also great when you are trying to translate SAS code into R. Many SAS DATA STEP operations can be mapped over directly to the R `datastep()` function.

Here are some of the features supported by the R `datastep()` function:

- Keep, Drop, and Rename
- Setting of Labels, Formats, and Attributes
- Retain and Calculate
- By Group Processing
- Output and Delete
- Set and Merge Operations

The bulk of this paper will examine the capabilities of the R `datastep`. But first, let's look at a simple example.

SIMPLE DATA STEP

The R `datastep()` function has only two required parameters: *data* and *steps*. The *data* parameter is the first parameter, and contains the input data to the `datastep`. The *steps* parameter is the second, and contains the code that will run for each row of the data. Output data is returned from the function. Here is an example showing these basic operations:

```
library(libr)

# Create sample data
dat <- read.table(header = TRUE, text = '
Name    Sex Age Height Weight
Alfred  M   14   69.0  112.5
Alice   F   13   56.5   84.0
Barbara F   13   65.3   98.0
Carol   F   14   62.8  102.5
```

```

Henry    M   14   63.5  102.5
James    M   12   57.3   83.0
Jane     F   12   59.8   84.5
Janet    F   15   62.5  112.5
Jeffrey  M   13   62.5   84.0')

# Calculate BMI
res <- datastep(dat, {

    BMI <- Weight / (Height ^ 2) * 703

})

# View results
res
#      Name Sex Age Height Weight      BMI
# 1  Alfred  M  14   69.0  112.5 16.61153
# 2   Alice  F  13   56.5   84.0 18.49855
# 3 Barbara  F  13   65.3   98.0 16.15679
# 4   Carol  F  14   62.8  102.5 18.27090
# 5   Henry  M  14   63.5  102.5 17.87030
# 6   James  M  12   57.3   83.0 17.77150
# 7    Jane  F  12   59.8   84.5 16.61153
# 8   Janet  F  15   62.5  112.5 20.24640
# 9 Jeffrey  M  13   62.5   84.0 15.11731

```

In the above example, notice the following:

- The input data is passed in the first parameter position.
- The output data is returned from the function and stored in the variable `res`.
- The “steps” are passed in curly braces.
- The BMI is calculated using standard R syntax and added to the end of the dataset.
- All other variables are kept.

While the syntax of the R `datastep` is not identical to the SAS DATA STEP, the overall concept is very similar. Most importantly, a SAS programmer will be able to easily read and understand the above code once they are accustomed to the R syntax.

KEEP, DROP, AND RENAME

The parameters *keep*, *drop*, and *rename* allow you to manage variables on the output dataset. These parameters will be executed after the “steps”.

KEEP AND RENAME

Using the above sample data, now let’s modify some variables. Here is an example showing how to use the *keep* and *rename* parameters:

```

# Calculate BMI
res <- datastep(dat,
    keep = v(Name, Height, Weight, BMI),
    rename = c(Name = "Subject"),
    {
        BMI <- Weight / (Height ^ 2) * 703
    })

```

```
# View results
res
#   Subject Height Weight      BMI
# 1  Alfred   69.0  112.5 16.61153
# 2   Alice   56.5   84.0 18.49855
# 3 Barbara   65.3   98.0 16.15679
# 4   Carol   62.8  102.5 18.27090
# 5   Henry   63.5  102.5 17.87030
# 6   James   57.3   83.0 17.77150
# 7    Jane   59.8   84.5 16.61153
# 8   Janet   62.5  112.5 20.24640
# 9 Jeffrey   62.5   84.0 15.11731
```

Notice the following in the above example:

- Keep parameters can be passed unquoted using the `v()` function.
- Rename parameters are passed as a named vector, mapping the original variable name to the new variable name.
- The rename operation occurs after the keep operation.

DROP

Like the *keep* operation, the *drop* operation also works by passing in a vector of variable names. This operation removes variables from a dataset:

```
# Calculate BMI
res <- datastep(dat,
               drop = v(Sex, Age),
               {
                 BMI <- Weight / (Height ^ 2) * 703
               })

# View results
res
#   Name Height Weight      BMI
# 1  Alfred   69.0  112.5 16.61153
# 2   Alice   56.5   84.0 18.49855
# 3 Barbara   65.3   98.0 16.15679
# 4   Carol   62.8  102.5 18.27090
# 5   Henry   63.5  102.5 17.87030
# 6   James   57.3   83.0 17.77150
# 7    Jane   59.8   84.5 16.61153
# 8   Janet   62.5  112.5 20.24640
# 9 Jeffrey   62.5   84.0 15.11731
```

Remember that all R variable and function names are case sensitive.

SETTING OF LABELS, FORMATS, AND ATTRIBUTES

The `datastep()` function allows you to set labels, formats, and other attributes, just like SAS.

LABEL AND FORMAT PARAMETERS

The *label* and *format* parameters are available to set variable labels and formats. The variable labels can be seen when opening a dataset in the RStudio® viewer. Variable formats are provided by the [fmtr](#) package. This package gives you the capability to attach formats to variables in the same manner as SAS. The format is applied using the [fdata\(\)](#) function. Let's see how this works:

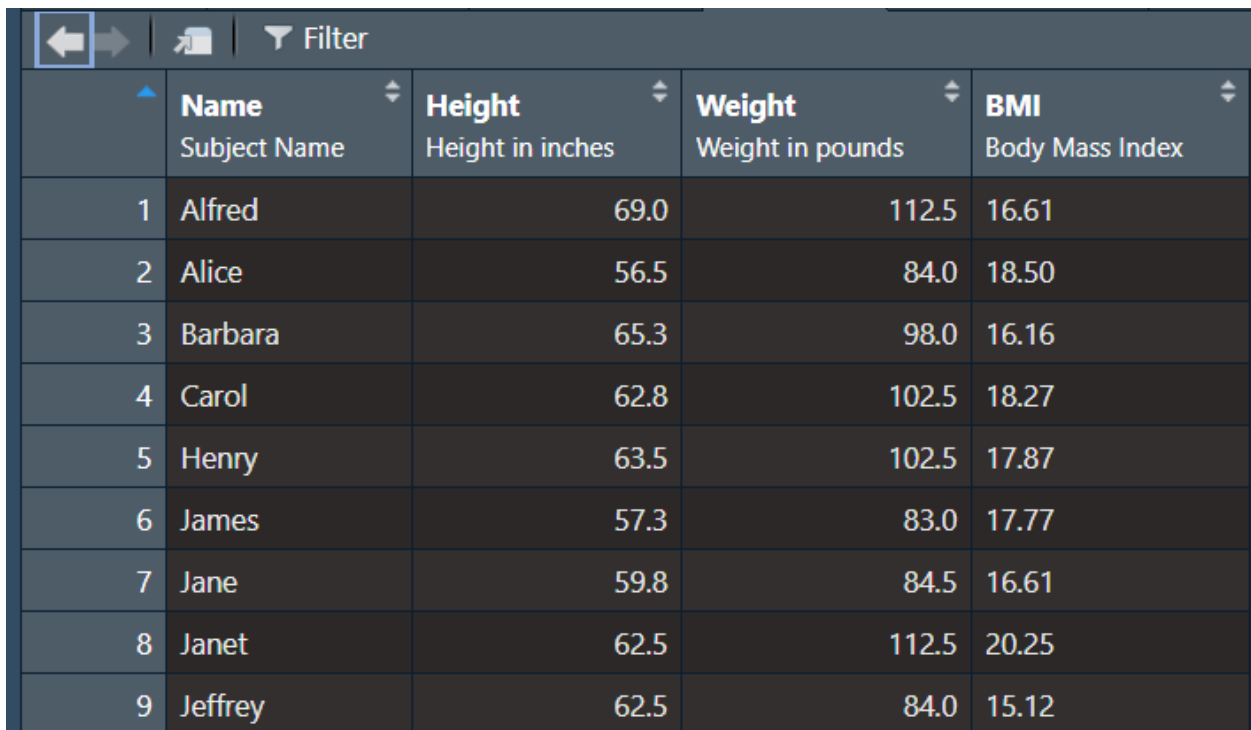
```
library(fmtr)

# Calculate BMI
res <- datastep(dat,
  keep = v(Name, Height, Weight, BMI),
  label = c(Name = "Subject Name",
            Height = "Height in inches",
            Weight = "Weight in pounds",
            BMI = "Body Mass Index"),
  format = list(BMI = "%.2f"),
  {
    BMI <- Weight / (Height ^ 2) * 703
  })

# Apply formats
res2 <- fdata(res)

# View results in Viewer
View(res2)
```

When the above code is executed, the following will appear in the RStudio viewer. Note the appearance of labels under the variable name. Also note the BMI variable is now formatted to two decimal places.



	Name Subject Name	Height Height in inches	Weight Weight in pounds	BMI Body Mass Index
1	Alfred	69.0	112.5	16.61
2	Alice	56.5	84.0	18.50
3	Barbara	65.3	98.0	16.16
4	Carol	62.8	102.5	18.27
5	Henry	63.5	102.5	17.87
6	James	57.3	83.0	17.77
7	Jane	59.8	84.5	16.61
8	Janet	62.5	112.5	20.25
9	Jeffrey	62.5	84.0	15.12

See the [fmtr](#) package documentation for more information about formatting in R.

DATASTEP ATTRIBUTES

The above labels and formats can also be assigned using the *attrib* parameter. This parameter takes a list of [dsattr\(\)](#) objects. A “dsattr” is a dataset attribute. This object has parameters to pass several variable attributes, such as label, format, default value, width, and justification. The next example will output the same results as the previous example, but using the *attrib* parameter instead.

```
library(fmtr)

# Calculate BMI
res <- datastep(dat,
  keep = v(Name, Height, Weight, BMI),
  attrib = list(Name = dsattr(label = "Subject Name"),
    Height = dsattr(label = "Height in inches"),
    Weight = dsattr(label = "Weight in pounds"),
    BMI = dsattr(label = "Body Mass Index",
      format = "%.2f")),
  {
    BMI <- Weight / (Height ^ 2) * 703
  })

# Apply formats
res2 <- fdata(res)

# View results in Viewer
View(res2)
```

As you can see, the output using the *attrib* parameter is the same as using the *label* and *format* parameters:

	Name Subject Name	Height Height in inches	Weight Weight in pounds	BMI Body Mass Index
1	Alfred	69.0	112.5	16.61
2	Alice	56.5	84.0	18.50
3	Barbara	65.3	98.0	16.16
4	Carol	62.8	102.5	18.27
5	Henry	63.5	102.5	17.87
6	James	57.3	83.0	17.77
7	Jane	59.8	84.5	16.61
8	Janet	62.5	112.5	20.25
9	Jeffrey	62.5	84.0	15.12

RETAIN AND CALCULATE

RETAIN

The *retain* parameter is one of the most useful features of a datastep. This parameter sets the initial value of the retain variable to the value of the previous row. This behavior allows you to calculate values across rows.

For instance, here is how to calculate a cumulative sum using the retain feature. Observe that the retain variables are passed as a list of variable names and initial values:

```
# Calculate Cumulative Height
res <- datastep(dat,
  drop = v(Sex, Age, Weight),
  retain = list(Total_Height = 0),
  {
    Total_Height <- Total_Height + Height
  })

# View results
res
#      Name Height Total_Height
# 1  Alfred   69.0         69.0
# 2   Alice   56.5        125.5
# 3 Barbara   65.3        190.8
# 4   Carol   62.8        253.6
# 5   Henry   63.5        317.1
# 6   James   57.3        374.4
# 7    Jane   59.8        434.2
# 8   Janet   62.5        496.7
# 9 Jeffrey   62.5        559.2
```

CALCULATE

The *calculate* parameter can be used to perform summary operations on all rows in the dataset. The *calculate* parameter is passed as set of R statements in curly braces, similar to the *step* parameter. The difference is that the calculate code runs first, and any variables created in the calculate block are then passed to the *steps*. This feature allows you to do things like calculate a sum or average, and then use that value to make decisions in your datastep.

Here is an example showing how to calculate the mean height, and then determine for each subject whether they are above or below the mean. This example also shows how to use an if/then/else conditional inside a datastep:

```
# Calculate Mean Height and Height Category
res <- datastep(dat,
  drop = v(Sex, Age, Weight),
  calculate = {
    Mean_Height <- mean(Height)
  },
  {
    if (Height > Mean_Height) {
      Height_Category <- "Above"
    }
  })
```

```

    } else if (Height < Mean_Height) {

      Height_Category <- "Below"

    } else {

      Height_Category <- "Equal"

    }

  })

# View results
res
#   Name Height Mean_Height Height_Category
# 1 Alfred  69.0    62.13333         Above
# 2  Alice  56.5    62.13333         Below
# 3 Barbara 65.3    62.13333         Above
# 4  Carol  62.8    62.13333         Above
# 5  Henry  63.5    62.13333         Above
# 6  James  57.3    62.13333         Below
# 7   Jane  59.8    62.13333         Below
# 8  Janet  62.5    62.13333         Above
# 9 Jeffrey 62.5    62.13333         Above

```

BY GROUP PROCESSING

By group processing is another very useful feature of the `datastep`. In the R `datastep()` function, the *by* group variables are identified on the *by* parameter. If there are more than one *by* variables, they can be passed in a vector. Just like the SAS `datastep`, `first.` and `last.` variables are created automatically, and can be used inside the data steps to make decisions or set values.

Here is an example using the `Height_Category` variable from the previous example, and identifying group boundaries for this variable:

```

# Sort data
dat2 <- sort(res, by = "Height_Category")

# Identify group boundaries
res2 <- datastep(dat2,
  by = "Height_Category",
  {
    if (first.Height_Category) {

      Boundary <- "First"

    } else if (last.Height_Category) {

      Boundary <- "Last"

    } else {

      Boundary <- "-"

    }
  })

```

```
# View results
res2
#      Name Height Mean_Height Height_Category Boundary
# 1  Alfred   69.0    62.13333      Above      First
# 2 Barbara   65.3    62.13333      Above        -
# 3  Carol    62.8    62.13333      Above        -
# 4  Henry    63.5    62.13333      Above        -
# 5  Janet    62.5    62.13333      Above        -
# 6 Jeffrey   62.5    62.13333      Above      Last
# 7  Alice    56.5    62.13333      Below      First
# 8  James    57.3    62.13333      Below        -
# 9   Jane    59.8    62.13333      Below      Last
```

OUTPUT AND DELETE

OUTPUT

Similar to SAS, the `output()` function will send the current row to the output dataset. This function may be used conditionally, so that only some of the rows are emitted to the final dataset.

The following example also picks up from the sorted `Height_Category` data generated above. Here we will use the `output()` function to output only the first and last rows in the group:

```
# Identify group boundaries
res2 <- datastep(dat2,
  by = "Height_Category",
  {
    if (first.Height_Category) {
      Boundary <- "First"
      output()
    } else if (last.Height_Category) {
      Boundary <- "Last"
      output()
    }
  })

# View results
res2
#      Name Height Mean_Height Height_Category Boundary
# 1  Alfred   69.0    62.13333      Above      First
# 2 Jeffrey   62.5    62.13333      Above      Last
# 3  Alice    56.5    62.13333      Below      First
# 4   Jane    59.8    62.13333      Below      Last
```

DELETE

The `delete()` function is the opposite of the `output()` function. Instead of sending the current row to the output dataset, the `delete()` function will remove it. Here is an example that produces the same output as above, but using the `delete()` function instead:

```

# Identify group boundaries
res2 <- datastep(dat2,
  by = "Height_Category",
  {
    if (first.Height_Category) {
      Boundary <- "First"
    } else if (last.Height_Category) {
      Boundary <- "Last"
    } else {
      delete()
    }
  })

# View results
res2
#      Name Height Mean_Height Height_Category Boundary
# 1  Alfred   69.0    62.13333      Above      First
# 2  Jeffrey   62.5    62.13333      Above      Last
# 3   Alice   56.5    62.13333      Below      First
# 4    Jane   59.8    62.13333      Below      Last

```

SET AND MERGE OPERATIONS

Set and Merge operations allow you to join two datasets together. The *set* operation stacks or unions two datasets on top of one another. The *merge* operation performs a join.

SET

The *set* parameter on the `datastep()` function accepts one or more datasets to append to the data passed on the *data* parameter.

To illustrate this operation, we will first split our dataset into two datasets. To split the dataset, we will use the `output()` function and the `n.` automatic variable. The `n.` automatic variable contains the current row number, and is similar to the `_n_` automatic variable in SAS:

```

# Create dataset 1
dat1 <- datastep(dat,
  keep = v(Name, Height, Weight),
  {
    if (n. < 5)
      output()
  })

# View results
dat1
#      Name Height Weight
# 1  Alfred   69.0   112.5
# 2   Alice   56.5    84.0
# 3 Barbara   65.3    98.0
# 4   Carol   62.8   102.5

```

```

# Create dataset 2
dat2 <- datastep(dat,
                keep = v(Name, Height, Weight),
                {
                  if (n. > 5)
                    output()
                })

# View results
#      Name Height Weight
# 1   James   57.3   83.0
# 2    Jane   59.8   84.5
# 3   Janet   62.5  112.5
# 4 Jeffrey   62.5   84.0

```

Now we can append the two datasets using a *set* operation:

```

# Append datasets
res <- datastep(dat1, set = dat2, {})

# View results
res
#      Name Height Weight
# 1  Alfred   69.0  112.5
# 2   Alice   56.5   84.0
# 3 Barbara   65.3   98.0
# 4   Carol   62.8  102.5
# 5   James   57.3   83.0
# 6    Jane   59.8   84.5
# 7   Janet   62.5  112.5
# 8 Jeffrey   62.5   84.0

```

In the example above, note that if the *datastep* has no steps, you must still pass empty braces (`{}`). The empty braces tell the function that there are no steps to execute.

MERGE

Dataset joins can be performed using the *merge*, *merge_by*, and *merge_in* parameters. The *merge* parameter identifies the dataset to join. The *merge_by* parameter identifies the variables to join by. Finally, the *merge_in* parameter names a set of indicator variables for the merge. These merge parameters work very similar to the SAS equivalents.

To see how the parameters operate, we must first create some sample datasets to merge:

```

# Create dataset A
datA <- datastep(dat1,
                keep = v(Name, Height),
                {
                  if (Name == "Alice")
                    delete()
                })

# View results
datA
#      Name Height
# 1  Alfred   69.0
# 2 Barbara   65.3
# 3   Carol   62.8

```

```

# Create dataset B
datB <- datastep(dat1,
  keep = v(Name, Weight),
  {
    if (Name == "Carol")
      delete()
  })

# View results
datB
#      Name Weight
# 1  Alfred  112.5
# 2   Alice   84.0
# 3 Barbara   98.0

```

Now that we have some data, let's look at some simple joins. The below example shows an inner join and a left join. You can see that the *merge_in* flags have been created automatically, and can be used to control which records are output:

```

# Inner Join
res <- datastep(datA,
  merge = datB,
  merge_by = c("Name" = "Name"),
  merge_in = c("inA", "inB"),
  drop = c("inA", "inB"),
  {
    if (inA & inB)
      output()
  })

```

```

# View results
res
#      Name Height Weight
# 1  Alfred   69.0  112.5
# 2 Barbara   65.3   98.0

```

```

# Left Join
res <- datastep(datA,
  merge = datB,
  merge_by = c("Name" = "Name"),
  merge_in = c("inA", "inB"),
  drop = c("inA", "inB"),
  {
    if (inA)
      output()
  })

```

```

# View results
res
#      Name Height Weight
# 1  Alfred   69.0  112.5
# 2 Barbara   65.3   98.0
# 3   Carol   62.8    NA

```

The NA value in the *Weight* column above is due to the left join, as the "Carol" row did not exist in the *datB* dataset.

For additional information, full documentation on the `datastep()` function can be found on the **libr** package web site at <https://libr.r-sassy.org/reference/datastep.html>.

PERFORMANCE CONSIDERATIONS

Note that the R `datastep()` function is slow compared to SAS. Take care when using the function with large datasets. When possible, subset the data first, before sending into the `datastep`. Try to reduce the data to only those rows and columns needed for the `datastep` operation. The Base R `subset()` function can be useful for this purpose. If performance is still too slow, use other Base R or Tidyverse functions to perform the desired manipulation.

CONCLUSION

The `datastep()` function from the **libr** R package contains many of the same features as the SAS DATA STEP. For programmers coming to R from SAS, this function can make working in R much easier. The function performs stepwise, row-by-row processing of data, and allows you to alter the dataset on the fly. It provides a conceptual understanding of the data that is often missing in other R functions. The `datastep` code is also very easy to read and understand. Combining the features described above gives you a very useful tool for manipulating your data in R.

REFERENCES

Bosak D (2023). *Datastep Operations*, Accessed August 9, 2024. <https://libr.r-sassy.org/articles/libr-datastep.html>

Bosak D (2023). *An Overview of the SASSY System*, WUSS Paper 185-2023, <https://www.lexjansen.com/wuss/2023/WUSS-2023-Paper-185.pdf>

ACKNOWLEDGMENTS

During development of the `datastep()` function, several people have provided ideas and encouragement. I'd like to thank Brian Varney, Duong Tran, and Kevin Kramer for their contributions.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

David J. Bosak
Archytas Clinical Solutions
dbosak01@gmail.com
www.r-sassy.org

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.