SESUG Paper 215-2023

Methods and Tools for Publishing at SESUG and Beyond-Advanced Techniques

Jonathan Duggins, North Carolina State University
James Blum, University of North Carolina Wilmington

Abstract

We add automation methods for including SAS code and output by building on "Methods and Tools for Publishing at SESUG and Beyond—Basic Techniques." We provide and demonstrate SAS programs for converting .SAS files to .TEX files. As the ODS TAGSETS.COLORLATEX command is pre-production and has limited features, we provide and demonstrate SAS code to modify sizing for tabular output objects based on corresponding RTF sizing. We combine SAS code and output building strategies with LaTeX tools for automatic insertion of the converted SAS code and SAS output into the LaTeX document. All LaTeX implementations are covered using the Overleaf platform, but other LaTeX environments work as well.

Introduction

In this paper, we build on the LaTeX and SAS tools presented in "Methods and Tools for Publishing at SESUG and Beyond – Basic Techniques" (hereafter referred to simply as Basic Techniques). As in Basic Techniques, to get the most out of this paper, it is necessary to be familiar with LaTeX – including loading packages, managing package options, and compiling documents. For readers just getting started with LaTeX, we suggest starting with the LaTeX wikibook: https://en.wikibooks.org/wiki/LaTeX. While we focus on publishing a SESUG paper, the techniques we introduce can be extended to other types of articles and even to books. This paper presents methods and tools, both in LaTeX and SAS, for automating much of the publication process to help ensure that code, call outs, and output objects such as tables and graphs are synchronously updated without relying exclusively on copy and paste or manual edits.

As with the Basic Techniques paper, this is a paper about how to write a paper and, as such, our structure for it is somewhat unusual because the final output is a complete paper in PDF form rather than a single table or graph. The sample paper is given in Appendix A – you should review it now to provide context for the following sections which provide the LaTeX and SAS code necessary to produce it. The sample paper and all associated files to produce it can be downloaded from https://jonathanduggins.com/conferences and additional information on setting up this project in Overleaf are provided in Appendix B for those who are not familiar with Overleaf.

We include our style files along with the sample paper as part of the download and have taken care to ensure you can use the style files with minimal, if any, modifications. However, we do discuss many of the easily accessible modifications so that you can adapt them to your projects if you need to do so. The discussion that follows is designed to match the order of the sample paper where possible.

Automatic Insertion of SAS Code Into Program Boxes

This section provides an overview of the steps we take to automate the inclusion of SAS code in the sample paper. LaTeX Code 1 shows the program box code we used to generate Program 1 in the sample paper. The red arrows indicate where wrapping has occurred to fit the LaTeX code into the box. In Overleaf, long code lines are automatically wrapped, and this wrapping changes dynamically as you resize the window (try it with your downloaded sample file). It is not advisable to insert breaks manually–LaTeX does allow line breaks inside a code line in some positions, but not in all positions.

LaTeX Code 1: Automatically Inserting Program 1

```
\begin{Program}{Using PROC MEANS to Compute Quartiles}{Means}❶
  \lstinputlisting❷[linerange=StartCode.Means-EndCode, language=SAS,
     ↪ includerangemarker=false, escapeinside=`~]{Code/Sample--
     ↪ AdvancedCode.tex}❸
\end{Program}
```

❶ The program box is built as in Basic Techniques: Begin the user-defined Program environment then provide a title for the box in the second argument and a label in the third argument.

❷ The LSTINPUTLISTING command is used to insert lines of code from an external file.

❸ LSTINPUTLISTING depends on several arguments to correctly insert the desired information from the provided external file. Details on these options are included in the sections below.

Of particular interest in LaTeX Code 1 is the fact that no SAS code is present; in fact, the LSTINPUTLISTING command is the only code between \Begin{Program} and \End{Program}. In Basic Techniques, we use the LSTLISTING command from the Listings package, which is a verbatim environment for typesetting code blocks in your paper. This requires the manual inclusion of the code – typically by copying and pasting the code from a SAS program. In contrast, the LSTINPUTLISTING command, also a verbatim environment and part of the Listings package, allows you to insert text from another file between designated stopping and starting positions. We include the same SAS code in the sample paper automatically by adding markers to our SAS program and then post-processing the SAS code to make it suitable for publishing in the LaTeX environment.

LSTINPUTLISTING Command Details in LaTeX Code 1

LaTeX Code 1 shows the basic statements necessary to automatically include lines of code from a source file. LaTeX Code 2 focuses on the LSTINTPUTLISTING command and its options.

LaTeX Code 2: LSTINPUTLISTING Option Details

```
\lstinputlisting[linerange=StartCode.Means-EndCode❶, language=SAS❷,
   ↪ includerangemarker=false❸, escapeinside=`~❹]{Code/Sample--
   ↪ AdvancedCode.tex❺}
```

❶ The LineRange option determines the lines of source code to insert into the LaTeX document. Multiple methods are available for determining the range – we use a named range.

❷ Setting the source code language to SAS leverages the built-in syntax highlighting provided by the Listing package. Currently this is limited for SAS, but may improve in future releases of the package.

❸ By default the markers that indicate the range of lines are included in the display text. Setting IncludeRangeMarker to FALSE prevents this behavior.

❹ As seen in Basic Techniques, the EscapeInside option sets starting and ending characters that wrap sections of text you do want to be interpreted by the LaTeX compiler.

❺ The source code is a required argument and must be included in your OverLeaf project. Here the file is in the Code folder. Note OverLeaf uses forward slashes in the Linux/Unix directory style.

As described in ❶, the LineRange option allows you to specify the set of lines in your source code – the file you identify in ❺. As a defensive programming tactic, we use named ranges rather than depending on line numbers. As code is edited, added, or removed – and similarly, call outs may be edited/added/removed – the line numbers of the desired code section will likely change. By using named ranges, the information provided to the LineRange option does not need to change as updates occur in the source file. However, this requires you to place the appropriate markers in your source file. As you proceed through the examples, you will see that we use a standardized set of markers to simplify the process and make it more robust. In LaTeX Code 2 you can see that we use a two-level name for the beginning of the range: the prefix StartCode is always used to identify the beginning of a section of code and a unique suffix (Means

for this case) distinguishes between specific sections of code in the full SAS code file. This suffix is also chosen to match the label given to the Program box, which is also Means as shown in LaTeX Code 1. The closing marker is simply EndCode in all cases, no distinction among code blocks is needed or desired.

Post-Processing SAS Code to Generate LaTeX Code

For the LSTINPUTLISTING command to import the code properly, we have to add range markers to our SAS code. We also choose to annotate the positions where call outs are placed in our SAS code. Once these mark up elements are added to our SAS code, it must be post-processed to produce the source TEX file referenced in the LSTINPUTLISTING command in LaTeX Code 2. In this section, we show the original SAS code and highlight the added markup. Then we show the related portions of the post-processing code. SAS Program 1 shows the actual SAS code that is post-processed to produce Program 1 in the sample paper.

SAS Program 1: Marking Up SAS Code for Program 1

```
*StartCode.Means;❶
ods select none;/*1*/❷
proc means data=sashelp.cars;
  where type not in ('Hybrid','Truck') and origin ne 'Asia';/*2*/❷
  class origin type;/*3*/❷
  var mpg:;/*4*/❷
  output out=summary q1=City_Q1 Highway_Q1 q3=City_Q3 Highway_Q3;/*5*/❷
run;

ods select all;
proc print data=summary;
run;
*EndCode;❸
```

❶ This marker indicates the beginning of the code we want included in the Program box. Post-processing is used to convert it to the marker expected by LSTINPUTLISTING.

❷ We mark desired locations for call outs with a comment indicating the appropriate number. Post-processing is used to convert to actual call out code.

❸ This marker indicates the end of the code we want included in the Program box. Post-processing is used to convert it to the marker expected by LSTINPUTLISTING.

In the first marker, note the use of the term Means – this is the unique suffix mentioned following LaTeX Code 2. However, you can see that the marker used here also includes an asterisk and semicolon, as they are both necessary for this style of SAS comment. Post-processing is used to remove these two characters when writing this line, and the line including the closing marker, to the resulting TEX file used as the source code in LaTeX Code 2.

To post-process the code shown in SAS Program 1, it is necessary to scan each line and make the desired adjustments. We employ a DATA step to carry out all SAS code post-processing and conversion to TEX. The entire DATA step appears complex when examined all at once, so SAS Program 2 focuses on the portion designed to post-process the code between the StartCode and EndCode markers.

SAS Program 2: Post-Processing Excerpt: Refining our PROC MEANS Code

```
data _null_; ❶
  length LineOut $ 2000;
  infile "&CodeFile..sas" truncover; ❷
  input @1 CodeLine $char2000.; ❷

  if anyalpha(CodeLine) gt 0 then do;❸
    if index(lowcase(CodeLine),'*startcode.') ne 0 then do;❹
        LineOut = compress(codeline,'*;');❺
        file "&CodeFile.Code.tex"; ❻
        LineOutLength=Length(LineOut); ❻
        put LineOut $varying2000. LineOutLength; ❻
        do until(lowcase(CodeLine) eq '*endcode;');❼
          infile "&CodeFile..sas" truncover;
          input @1 CodeLine $char2000.;
          if lowcase(codeline) eq '*endcode;'
               then LineOut = compress(CodeLine,'*;');❺
            else LineOut =
               tranwrd(tranwrd(CodeLine,'/*','`\CallOut{'),'*/','}~'); ❽
          file "&CodeFile.Code.tex";❻
          LineOutLength=Length(LineOut);❻
          put LineOut $varying2000. LineOutLength;❻
        end; /**end the DO UNTIL loop**/
    end; /**end starting marker DO group**/
    ❾
  end; /**end DO group with ANYALPHA**/
run;
```

❶ We use a _NULL_ DATA step since the only intent is to create a TEX file.

❷ Read a line from the original SAS code, using TRUNCOVER to avoid reading the next line when reaching past the end of lines and using the $CHAR informat to preserve leading spaces that may be present in the code.

❸ Post-processing is triggered for lines that contain special markers (like StartCode), so lines without characters can be skipped.

❹ When a StartCode marker is identified, we begin post-processing all lines in that code block.

❺ Remove the asterisk and semicolon from the StartCode and EndCode markers in the SAS program so that they have the form expected by the LSTINPUTLISTING command.

❻ For every line from the original SAS code, it is post-processed and stored in the DATA step variable LineOut and then written to the TEX file. For each line the length is determined for use with the $VARYING format so that spaces are not added to the end of the line unnecessarily.

❼ Continue reading lines of SAS code from the original program until you encounter the EndCode marker.

❽ Any line that contains a call out indicator – we use /*N*/, with N a whole number between 1 and 10 – is modified via the TRANWRD function to become the LaTeX command \CallOut{N}. Note the characters ' and ~ are prepended and appended to the CallOut command because they are the EscapeInside characters designated in our LSTINPUTLISTING command.

❾ There are additional lines in our post-processing DATA step for other items like call out lists. They are not shown here, but are discussed in later sections when they are demonstrated.

Note that unlike the starting and ending markers that have specific text to identify those lines for post-processing, the call out tags are simply the standard /* and */ tags used for comments in SAS. As a

result, you cannot use those tags to include a comment in your program. If you attempt to do so, the post-processing inserts the comment text as the argument to the CallOut function, leading to an error from the LaTeX compiler. Since you have call outs and call out lists, it is expected that you will not need such comments in your code in the paper. If you do want to comment your code in that style, you will need to modify the tags used to indicate a call out in your SAS code, and modify the post-processing DATA step to use those tags.

The result of applying the post-processing code in SAS Program 2 to the original SAS code shown in SAS Program 1 is a TEX file that must be uploaded to Overleaf (or placed in whatever LaTeX environment you choose to use). The LSTINPUTLISTING command is then given with the chosen range markers and file reference, as shown in LaTeX Programs 1 and 2. For brevity, LaTeX Program 3 shows only the first few lines of post-processed SAS code since the remaining lines are similarly modified. The full block of LaTeX code is available as part of the project files you downloaded. Note that the StartCode.Means marker is now free of SAS elements like the asterisk and semicolon and the SAS comment indicating a call out has been replaced with the corresponding LaTeX command.

LaTeX Code 3: Post-Processed Means Code - Partial

```
StartCode.Means
ods select none;`\CallOut{1}~
```

The post-processing code in SAS Program 2 provides you with the ability to turn your marked up SAS program into a TEX file that contains the markers expected by the LSTINPUTLISTING command. While this process may seem overwhelming at first, recall the post-processing we provided is designed to be used without need for any changes. If you choose to use the StartCode and EndCode terms in your markers like we do, then you can use this program exactly as presented to post-process your SAS programs. If you choose to use different markers, ensure the INDEX function and DO UNTIL loop are modified to scan for your choices. You only need to make these edits once during your publication process. Further, the same post-processing code can be used across all of your publishing projects.

In the sections below, we take a look at the other post-processing needs: inserting call out lists, inserting tables, and inserting graphs. As part of the conclusion, we provide a high-level view of the process including a list of steps to get you started with automating your publishing process.

Inserting A Call Out List from SAS Code

To add the call out list into the paper, we could type it directly into the LaTeX editor as shown in Basic Techniques; however, we consider a different approach here. We place the call out list into the SAS program as a comment following the EndCode marker. This permits the call out list to be updated as the code is edited and/or any call out markers are added or changed. Inclusion of the call out lists in the final paper follows a process similar to the one for inserting code. The main difference arises because the call out lists are placed outside Program boxes in a style following the LaTeX typesetting; therefore, we do not want use a verbatim environment like LSTINPUTLISTING.

To bring in text from another file that we expect to use LaTeX typesetting, we use the CatchFileBetweenTags package. This package provides various commands for inserting text from a source TEX file into your current document, and we use the EXECUTEMETADATA command. SAS Program 3 shows the portion of the original SAS program where the call out list is written, with its opening and closing markers, that appears as the call out list following Program 1 in the sample paper.

SAS Program 3: Marking Up the Call Out List

```
❶

❷*%<*Enumerate.Means> ❸
1~❹ There is no need to see the standard output from the MEANS Procedure,❺
  so it is suppressed.
2~The WHERE statement reduces the data to the desired categories.
  It is necessary to do this for Type at this step so that unwanted
  observations do not affect the calculation of the statistic.
3~Origin is one of the panel variables, and Type is the charting variable
  within each panel, so stratification on each is necessary.
4~Recall, the : is a wildcard for an arbitrary suffix,
  so this includes both MPG variables.
5~In the OUTPUT statement, each statistic keyword has two variable names
  listed after it as there are two analysis variables in the VAR statement.
%</Enumerate.Means>;❸
```

❶ The code from SAS Program 1 appears at this point in the source code but, for brevity, is not repeated here.

❷ To include the call outs in the SAS code, the starting marker, call out list, and ending marker are all included in a single comment, beginning with the *, ending with ; – which are removed in post-processing.

❸ For the EXECUTEMETADATA command, the opening marker is of the form `%<*marker-name>`, with the ending marker similar, having * replaced by / (`%</marker-name>`). The marker-name must be unique within the file referenced.

❹ Rather than writing `\item` repetitively in the SAS code, where the command has no meaning anyway, we use the ~ to indicate the beginning of the item. Post-processing is used to replace the character with the necessary LaTeX code. The numbers are there for convenience when editing the SAS code, they are removed in post-processing.

❺ The wrapping here is not only for display purposes in this paper. To avoid long lines in the SAS code, line breaks are inserted to wrap the text in the SAS editor. LaTeX ignores a single return – it does not actually create a line break – so the call out is still rendered correctly.

Note that this is little extra work compared to including the call out list directly in the final paper. In either case the call out list must be typed out and placed in an enumerated list. Therefore, the choice is a matter of where you prefer to store and edit your lists – in the SAS program or in the LaTeX program.

When constructing our start and end markers, we use a two-level name with a prefix that is always used to identify the beginning of a call out list and unique suffix just as with the StartCode markers for our code sections. The prefix Enumerate highlights the intent of the marked up section and the suffix uses the consistent term – Means – that has appeared throughout this example to tie the various elements together. We prepend each call out with its number simply to make it easier to match call outs descriptions to the call out tag in the code itself. This helps ensure that the call out text is in the correct position in the list. SAS Program 4 shows the post-processing code we use to prepare the call out list for automatic insertion into the paper.

SAS Program 4: Post-Processing Excerpt: Preparing A Call Out List

```
data _null_;
  length LineOut $ 2000;
  infile "&CodeFile..sas" truncover;
  input @1 CodeLine $char2000.;

  if anyalpha(CodeLine) gt 0 then do; ❶

    ❷

      else if substr(CodeLine,1,4) eq '*%<*' and
              index(lowcase(CodeLine),'enumerate') ne 0 then do;❸
          LineOut = compress(substr(CodeLine,2),';');❹
          file "&CodeFile.Code.tex";
          LineOutLength=Length(LineOut);
          put LineOut $varying2000. LineOutLength;
          do until(substr(CodeLine,1,3) eq '%</');❺
            infile "&CodeFile..sas" truncover;
            input @1 CodeLine $char2000.;
            if substr(codeline,1,3) eq '%</'
                then LineOut = compress(CodeLine,';');❻
              else do;
                  position=index(Codeline,'~');❼
                  if position eq 0 then position = 1;❼
                  cText = substr(CodeLine,position); ❽
                  LineOut = strip(tranwrd(ctext,'~','\item '));❾
                end;
            file "&CodeFile.Code.tex";
            LineOutLength=Length(LineOut);
            put LineOut $varying2000. LineOutLength;
          end; /**end the DO UNTIL loop**/
        end; /**end starting marker DO group**/
    end; /**end DO group with ANYALPHA**/
run;
```

❶ Recall the post-processing is triggered for lines that contain special markers, so lines without characters can be skipped.

❷ The post-processing for handling code blocks shown in SAS Program 2 appears at this point in the SAS program but, for brevity, is not repeated here.

❸ Since the markers must appear in a comment, when the starting-specific marker sequence `*%<*` is identified in a line containing our chosen prefix (enumerate), we begin post-processing all lines in that code block.

❹ Remove only the initial asterisk and any semicolons from the first line so that it has the form expected by the EXECUTEMETADATA command.

❺ Continue reading lines of SAS code from the original program until you encounter the ending-specific marker sequence `%</`.

❻ Remove the semicolon from the final line in the call out block.

❼ Determine the position of our chosen character to indicate a new call out. For lines with the character, store its position in the string. If no special character is present, set the position to be 1.

❽ Store the text from the current call out starting from the previously defined value of Position. Note: in the code provided with the Overleaf project, this is included in the following line of SAS code. It is

placed on a separate line here to avoid multiple call outs in a single statement broken across lines.

❾ Replace the special character with the necessary LaTeX code to create a new item in the enumerated list.

Once you run the post-processing on your call out list, the result is LaTeX code ready for you to insert into your final document. LaTeX Code 4 shows the first few lines of the sanitized code for this PROC MEANS call out list. Note the first line matches the marker syntax expected by the EXECUTEMETADATA command and the second line shows the insertion of the `\item` command in place of the ~ that acted as a placeholder.

LaTeX Code 4: Result of Post-Processing the Call Out List

```
%<*Enumerate.Means>
\item There is no need to see the standard output from the MEANS Procedure,
so it is suppressed.
```

The final step is to insert this code into your LaTeX program for inclusion in your published document. This process is even simpler than the one discussed earlier for inserting code. LaTeX Code 5 shows the process – just place an EXECUTEMETADATA command inside the customized enumerated list environment.

LaTeX Code 5: Using the EXECUTEMETADATA Command

```
\begin{enumerateCallOut}❶
    \ExecuteMetaData[Code/Sample--AdvancedCode.tex❷]{Enumerate.Means}❸
\end{enumerateCallOut}
```

❶ The EnumerateCallout environment is included in the SESUGArcticleStyle style file we provide. This environment uses a standard enumerated list and modifies the items to use the appropriate character in place of the default Arabic numbers.

❷ Just as with the LSTINPUTLISTING command, provide the path to the source file that contains the LaTeX code you want to insert.

❸ Provide your selected marker for the code section. Ensure you only provide the marker name – the mark up characters `%<*` and `%</` must not be included here.

The result of executing the code shown in LaTeX Code 5 is the call out list shown after Sample Program 1. Of course, from only the published document it is indistinguishable from an enumerated list that was typed directly into the paper itself. As with the automatic insertion of the SAS code blocks, automatic inclusion of the call out lists may seem daunting due to the different SAS and LaTeX code pieces displayed in this section. However, keep in mind that the post-processing code only needs to be written at the outset and that we have provided a program that you should be able to use with minimal changes, if any.

As mentioned above, you have to write the call out list at some point, so including it in your SAS program does not affect the overhead for creating this list. Finally, the post-processing automatically prepares your LaTeX code for inclusion, so only the three lines to place the EXECUTEMETADATA command in an enumerated list need to appear in your final LaTeX program. As a benefit, having the call out list with the SAS code ensures that the SAS code retains this information as comments, and when the SAS code is updated, the call outs can be updated simultaneously and re-loaded into your LaTeX project. Once coupled with the automatic inclusion of output objects, discussed below, this automated workflow generally reduces project management time and decreases the number of mistakes in the published document.

Automatically Generating Output 1

This section discusses the automated inclusion of one of the most important elements – tabular output objects generated by example code. Ideally, placing a table into your LaTeX program would be as easy as using the ODS TAGSETS.COLORLATEX destination and then using EXECUTEMETADATA to insert those

rows into your final LaTeX program. However, there are a few considerations that make this more challenging initially when compared to inserting call out lists.

The primary concerns with generating tabular output and inserting it automatically into your published work are related to the size of the resulting tables. The number of rows in the table may need to be limited or the column widths may need to be adjusted. Vertical sizing issues may arise if you want to provide code, such as PROC PRINT or PROC FREQ, that would produce a long output object but only want to show a portion of that object. We employ a macro variable to control the number of rows displayed in the included output. By doing so, readers have the benefit of having the correct and complete code but without an oversized table detracting from the main points in the paper.

The issue of column widths is more complex because widths are determined automatically as part of the output process controlled by ODS. Since the TAGSETS.COLORLATEX destination was pre-production and is currently not being maintained, we apply modifications to ensure the final tables wrap text properly and are displayed with appropriate widths. As the family of LaTeX destinations for ODS continue to develop, we hope SAS makes the need for these modifications obsolete. Because the modifications related to the table widths require at least some understanding of how various ODS destinations format the code that produces tables, we present an overview of the modifications that we choose to employ but do not examine the post-processing code in detail. The full post-processing code for these elements is included in the project download and you can apply them to your own projects without modification. SAS Program 5

SAS Program 5: Marking Up SAS Code to Generate Sample Output 1

```
%let OutputName = SESUG; ❶

❷

ods rtf file = "&OutputName.RawTables.rtf" ❷
        style = styles.test; ❷

ods tagsets.colorlatex ❸
      file = "&OutputName.RawTables.tex" (notop nobot) ❸
      stylesheet = "SAS.sty"; ❸

proc odstext;  ❹
  p "TableRows="; ❺
  p '%<*Means>'; ❻
run;

❼

proc odstext; ❽
  p '%</Means>';
run;

ods tagsets.colorlatex close;
ods rtf close;
```

❶ We use a macro variable to name output files. This simplifies the naming process for projects with multiple output files.

❷ We provide PROC TEMPLATE code for constructing a custom RTF style. The TEMPLATE code is not shown, but is applied in the ODS RTF statement.

❸ We use the TAGSETS.COLORLATEX destination as described in the Basic Techniques paper.

❹ PROC ODSTEXT inserts a text block into available ODS destinations. The P statement begins a paragraph.

5. TableRows is the name of our macro variable that controls how many non-header rows to display in the output object. Post-processing looks for this information and stores it in the macro variable.
6. This inserts the starting marker using the syntax expected by the EXECUTEMETADATA command. Note the use of our consistent label: Means.
7. The marked up SAS code to produce the output (shown in SAS Program 1) and the call out list (shown in SAS Program 3) are omitted here.
8. We again use PROC ODSTEXT to insert the necessary markup. Here we insert the closing tag for this example.

The use of PROC ODSTEXT to add elements such as the markers into the file is different than the earlier approaches that used comments. The difference is necessary since the file we are marking up in this case is the actual TEX output and not the SAS code. Since comments are not executed and produce no output, they would not appear in our TEX file. A tool like PROC ODSTEXT is thus necessary to allow SAS to deliver our markup with the rest of the output objects.

In addition to the TEX file, we generate an RTF file so that we can extract the table widths from the RTF code and use them in the post-processing of the TEX file. This is currently necessary because the LaTeX family of ODS destinations create tables using an HTML-like style that does not include text wrapping. We create custom column environments in the style file we provide with the project and use them along with the RTF-produced widths to generate a table suitable for printing. This post-processing is done via the code in OutputCleaning.sas; it uses the RTF and TEX files created in SAS Program 5 as input and produces a final set TEX file that includes the sanitized tables. During this process we also remove extraneous lines SAS adds to the original TEX file. These files are included in the project download so you can compare the TEX files before and after the post-processing. The LaTeX code necessary to insert the output looks similar to the insertion of the call out list since both use the EXECUTEMETADATA command. LaTeX Code 6 shows how to carry this out.

LaTeX Code 6: Including SAS Output to Produce Sample Output 1

```
\begin{Output}{\ref*{P:Means}: Data Set from PROC MEANS
        ↪OUTPUT Statement}{Means}
    \ExecuteMetaData[Output/Advanced/SESUGFinalTables.tex]{Means}
\end{Output}
```

It is important to understand that you may need to produce a table that is so wide that it wraps in the RTF – one observation requires more than one row in the table to display all requested columns. In this case, you will need to make decisions about how to display your table in LaTeX – the post-processing will not fix this issue.

Revisiting Sample Paper Output 2

In the Basic Techniques paper we show how to place call outs in an output object by manually inserting them in the table code. However, this conflicts with the concept of automation we are prioritizing in this paper. These manual call out insertions would need to be maintained since each time the SAS code generated a new TEX file, any previous manual changes would be lost. If manual changes are essential, then we suggest delivering those output objects to individual TEX files so that they do not need to be updated after each execution of your SAS program.

Additionally, note that When building Output 2 in the sample paper, only a partial listing is provided as the actual data set is a bit longer. In Basic Techniques, this is manually handled by deleting the unwanted rows from the TEX file before importing it via the INPUT command. However, now that we have an automated way to handle table length with post-processing there is no need to manually update the file to handle this issue either. Recall SAS Program 5 showed the PROC ODSTEXT step which included assigning a null value to TableRows since we display the full table in Output 1 in the sample paper. To ensure Output 2

from the sample paper shows only the first 12 rows, we set TableRows equal to 12 in the original SAS program. Post-Processing automatically displays the header along with the requested number of rows.

Conclusion

Automatically including SAS code, call outs, and tabular output objects enables a more seamless publication process by ensuring that all included elements are in sync. In this paper we demonstrate a process for integrating this level of automation in your workflow. In particular, the tools presented in this paper allow for the following approach.

1. Write your original SAS program including call out lists all necessary markup.
2. Run your original SAS program to generate your raw tables TEX file.
3. Run the post-processing programs:
   (a) Output post-processing to turn your raw tables TEX file into the final tables TEX file.
   (b) Code post-processing to turn your original SAS code into a TEX file.
4. Insert your content as needed into the TEX file containing your SESUG paper (or other document):
   - Use the LSTINPUTLISTING command to insert lines of code
   - Use the EXECUTEMETADATA command to insert call out lists or tabular output

We provide our post-processing files for you to use and, by design, they require minimal maintenance between projects. This reduces the time needed for project management, resulting in a more efficient process because your post-processing files are reusable across projects. SAS log content and figures can still be inserted as shown in the Basic Techniques paper. LaTeX offers a rich set of potential customizations – from creating your own functions to including entire packages – all of which can result in a more efficient and enjoyable publishing process. We encourage everyone to start with the provided files to see this workflow in action and then apply it to your next project.

References

Blum, J. and Duggins, J. 2023. “Methods and Tools for Publishing at SESUG and Beyond – Basic Techniques.” SESUG Proceedings

Recommended Reading

Wikipedia. 2023. “LaTeX Wikibook.” https://en.wikibooks.org/w/index.php?title=LaTeX&stableid=4246546

Contact Information

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

Jonathan Duggins

North Carolina State University

jwduggin@ncsu.edu

https://jonathanduggins.com/

Appendix A–Sample Paper

SESUG Paper xyz-2023

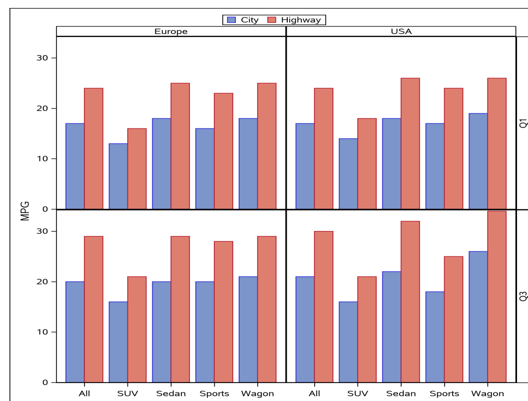
Sample Paper-Advanced Techniques

Jonathan Duggins, North Carolina State University
James Blum, University of North Carolina Wilmington

Introduction

Our "sample paper" goes through steps to make the panel graph shown in Graph 1 using the Cars data from the SASHELP library.

Graph 1: Desired Panel Graph



First, since the quartiles are included in the graph, the summary statistics need to be generated outside of PROC SGPANEL. In Constructing a Data Set of Statistics, the MEANS Procedure is used to create the summary statistics and store them in a SAS data set, which is then transformed to a suitable form in Constructing a Data Set of Statistics. Finally, the graph is created in Making the Panel Graph

Building the Summary Data

Several candidates for computing the needed quartiles are available, we choose to use PROC MEANS. No output data sets from PROC MEANS have a structure that fits the requirements for the panel graph, so some transformations are required.

Constructing a Data Set of Statistics

Program 1 uses PROC MEANS to get the first and third quartiles for city and highway MPG across the desired categories.

Program 1: Using PROC MEANS to Compute Quartiles

```
ods select none;❶
proc means data=sashelp.cars;
  where type not in ('Hybrid','Truck') and origin ne 'Asia';❷
  class origin type;❸
  var mpg:;❹
  output out=summary q1=City_Q1 Highway_Q1 q3=City_Q3 Highway_Q3;❺
run;

ods select all;
proc print data=summary;
run;
```

❶ There is no need to see the standard output from the MEANS Procedure, so it is suppressed.

❷ The WHERE statement reduces the data to the desired categories. It is necessary to do this for Type at this step so that unwanted observations do not affect the calculation of the statistic.

❸ Origin is one of the panel variables, and Type is the charting variable within each panel, so stratification on each is necessary.

❹ Recall, the : is a wildcard for an arbitrary suffix, so this includes both MPG variables.

❺ In the OUTPUT statement, each statistic keyword has two variable names listed after it as there are two analysis variables in the VAR statement.

Looking at Output 1, we can see that the data set generated by the OUTPUT statement includes marginal summaries not shown in the standard PROC MEANS output. These can be controlled by the WAYS statement; however, we will subset them during the transformation process. There are four variables containing the MPG quartiles, but SGPANEL graph uses one in the role of the RESPONSE= variable. The two different quartiles are values of a panel variable, and the city and highway categories are the group variable in the charts. Clearly, some transformation is required.

Output 1: Data Set from PROC MEANS OUTPUT Statement

Obs	Origin	Type	_TYPE_	_FREQ_	City_Q1	Highway_Q1	City_Q3	Highway_Q3
1			0	254	17	24	21	29
2		SUV	1	35	14	18	16	21
3		Sedan	1	168	18	25	21	30
4		Sports	1	32	16	23	19	27
5		Wagon	1	19	18	25	22	30
6	Europe		2	123	17	24	20	29
7	USA		2	131	17	24	21	30
8	Europe	SUV	3	10	13	16	16	21
9	Europe	Sedan	3	78	18	25	20	29
10	Europe	Sports	3	23	16	23	20	28
11	Europe	Wagon	3	12	18	25	21	29
12	USA	SUV	3	25	14	18	16	21
13	USA	Sedan	3	90	18	26	22	32
14	USA	Sports	3	9	17	24	18	25
15	USA	Wagon	3	7	19	26	26	34

Transforming the Data Set for Use in the Panel Graph

First, Program 2 uses PROC TRANSPOSE to create a data set with a single column of MPG quartiles.

Program 2: Transposing the Quartile Data

```
proc transpose data=summary
   out=summary2(drop = _type_ _label_❶ rename=(col1=MPG)❷);
  where not missing(origin);❸
  by _type_ origin type;❹
  var city: highway:;
run;

proc print data=summary2;
run;
```

❶ These variables of no use after this step, but the DROP= option must be here for both: _type_ is used in the BY statement, _label_ is created by PROC TRANSPOSE.

❷ This renaming is certainly not required, but provides a more intuitive name to the new column of MPG quartiles.

❸ Here, we remove the cases not needed for the graph.

❹ _type_ is the primary sorting variable, with the secondary and tertiary variables corresponding to the CLASS statement in Program 1.

Output 2: Transposed Quartile Data, Partial Listing

Obs	Origin	Type	_NAME_	MPG
1	Europe		City_Q1	17
2	Europe		City_Q3	20
3	Europe		Highway_Q1	24
4	Europe		Highway_Q3	29
5	USA		City_Q1	17
6	USA		City_Q3	21
7	USA		Highway_Q1	24
8	USA		Highway_Q3	30
9	Europe	SUV	City_Q1	13
10	Europe	SUV	City_Q3	16
11	Europe	SUV	Highway_Q1	16
12	Europe	SUV	Highway_Q3	21

Output 2 shows a partial printout of the transposed quartile data. Because Type is empty for the marginal statistics associated with Origin, values for the missing cases will be filled in to make an intuitive value for use in the chart. Additionally, the values of _NAME_need to be split into two components across the underscore. Generally, we would not advocate including an underscore in variable names; however, this is a useful exception. Program 3 finishes the transformation of the data; however, it does contain one fairly inconsequential logic error. Can you find it? The SAS log that follows may be helpful.

Program 3: Final Transformation of the Quartile Data

```
data summary2;
  set summary2;
  if missing(type) then type = 'All';
  Statistic = scan(_name_,2,'_');
  Category = scan(_name_,1,'_');
  drop _name_;
  length statistic category $8;
run;
```

Log 3: Final Transformation of the Quartile Data

```
174          data summary2;
175            set summary2;
176            if missing(type) then type = 'All';
177            Statistic = scan(_name_,2,'_');
178            Category = scan(_name_,1,'_');
179            drop _name_;
180            length statistic category $8;
WARNING: Length of character variable Statistic has already been set.
         Use the LENGTH statement as the very first statement in the
         DATA STEP to declare the length of a character variable.
WARNING: Length of character variable Category has already been set.
         Use the LENGTH statement as the very first statement in the
         DATA STEP to declare the length of a character variable.
181          run;

NOTE: There were 40 observations read from the data set WORK.SUMMARY2.
NOTE: The data set WORK.SUMMARY2 has 40 observations and 5 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

Making the Panel Graph

Now the data is in a form that will work well with PROC SGPANEL. Program 4 creates the same chart shown in Graph 1.

Program 4: Making the Panel Graph

```
proc sgpanel data=summary2;
  panelby origin statistic ❶ / layout=lattice novarname ❷;
  vbar type / group=category response=MPG groupdisplay=cluster;❸
  rowaxis label='MPG' offsetmin=0;❹
  colaxis display=(nolabel);❹
  keylegend / position=top title='';❹
run;
```

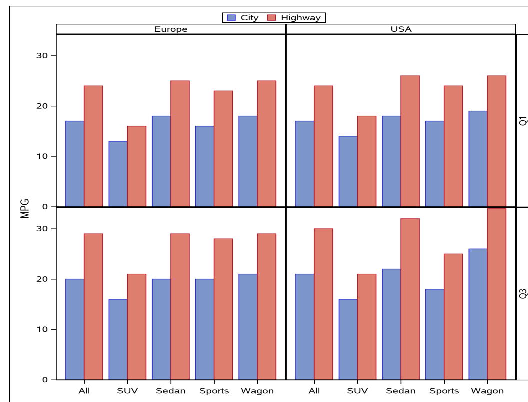
❶ Origin and Statistic are the two variables that define the four panels.

❷ The LATTICE layout puts the first variable on the columns, second on the rows. NOVARNAME suppresses the variable name in the panel labels.

❸ Each panel is a grouped bar chart: charting variable is Type, group variable is Category (City/Highway), and response is MPG.

❹ AXIS and KEYLEGEND statements are used to alter default labels and styles.

Output 4: The Panel Graph



And so ends our sample paper.

Return to the Full Paper Introduction

Appendix B–Downloading the Sample Paper and Setting Up Overleaf

When you download the sample file from Demo Project File, you are downloading a .zip file. Overleaf is equipped to create an entire project from an appropriate .zip file. First, select "Upload Project" from the drop-down menu given by the "New Project" button, as shown in the first panel of Figure 1. This pops up a box to select a .zip file from your computer (second panel)–navigate to the file you downloaded and select OK, or drag your file to the box. If you are successful, Overleaf will immediately open the project, and you will see an environment like the third panel of Figure 1.

Figure 1: Uploading a Project to Overleaf