PharmaSUG 2025 - Paper AP-097

Better, Faster, Stronger: A Case Study in Updating SAS Macro Code with Run-time Optimization in Mind

Valerie Finnemeyer, Center for Biostatistics in AIDS Research (CBAR); Harvard TH Chan School of Public Health

ABSTRACT

We typically write code for a specific purpose, then "macrotize" it for more general use. Over time, additional features are tacked onto the original code, turning it into "Frankencode" that functions, but not always optimally. Ideally, we could start over when our original macro code becomes unwieldy, building a new version of it from the ground up. But this may not be practical, and the resulting improvements may not justify the time we'd have to spend rewriting the code.

This paper describes a process for updating complex macros to reduce runtime. It then illustrates how to use this framework by applying it to a CBAR macro that performs and aggregates descriptive statistics from a CDISC ADaM BDS or ADSL dataset to create publication-ready tables. I describe the three-step process:

- brainstorming potential improvements,
- testing and assessing their impact, and
- deciding how to update the macro to minimize programmer effort and maximize user efficiency gain.

The CBAR macro I describe uses SAS code, but the framework applies to all programming languages.

INTRODUCTION

We routinely use macros to automate repetitive processes, saving coding and validation time. But we can't always build a macro that provides an optimal solution to a problem, even when following the best practices for code and macro design in the literature[2,3,6]. Optimizing macro function seems simple, but grows complicated as users add new features to existing macros and their scope expands incrementally. The more functional and complex a macro becomes, the harder it is to add new features to it or the more time-intensive it becomes to reconstruct it. Clunky macros may create bottlenecks and frustrate users, but redesigning a macro may take so long it would be infeasible. Fortunately, we can make smaller changes to macros that can reduce their runtimes without having to start over from scratch.

This paper presents a three-step process for updating macros that helps programmers identify high-impact, low-effort options for reducing macro runtime and determine if and how to implement them. Figure 1 outlines this process. The rest of this paper walks through its application to a case study in updating one of our macros. I describe each step in more detail and give examples of how it was implemented in our case study.

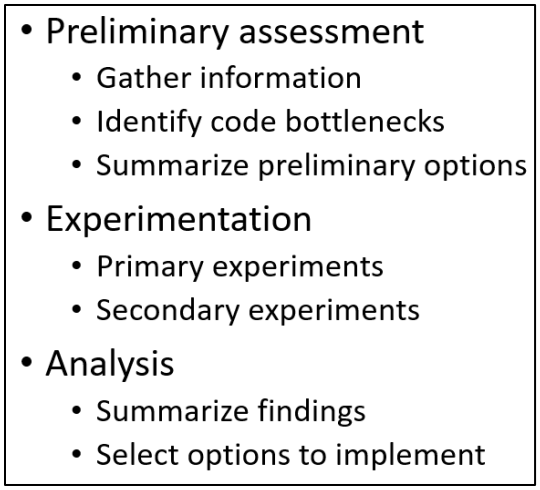

Figure 1. Macro optimization process

CASE STUDY OVERVIEW

This case study applies the macro optimization framework to the CBAR %BDSSTATS macro. This section briefly describes this macro and its general process flow to give context for the discussion of the update process.

MACRO PURPOSE

%BDSSTATS was created in 2017 to provide summary statistics for standard ADaM BDS and ADSL datasets. It combined features and code from two macros that processed non-CDISC (legacy) data and layered in handling of ADaM data structures and addressing CDISC-compliance (e.g. traceability). %BDSSTATS provides output to users in a format nearly ready to feed to a PROC REPORT to generate publication-ready tables.

MACRO STRUCTURE

Figure 2 shows the basic flow of %BDSSTATS. After the code parses the input list of requested parameters/variables and statistics, it loops over every parameter/variable to determine which PROCs to run to obtain all of the requested statistics. These PROCs then output all statistics that %BDSSTATS supports, eliminating output statistics the user didn't request for that parameter/variable. %BDSSTATS then combines all PROC output and other useful parameters that assist PROC REPORT, such as ordering variables, into one dataset for the user.

REASON FOR UPDATING %BDSSTATS MACRO

The %BDSSTATS macro is undergoing an update to include new features, including statistical procedures/outputs not originally supported. In defining the scope of the update, the study programmers/statisticians asked the macro programming team if runtime could be improved. As we took on this challenge, a key consideration was to balance the expected payoff with the effort involved.

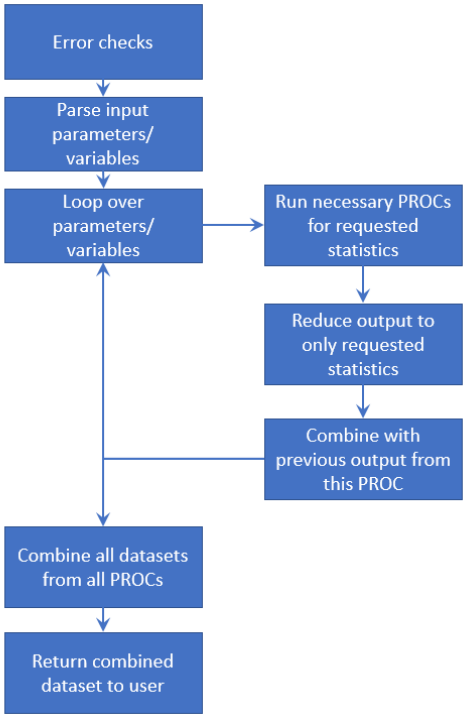

Figure 2. Overview of %BDSSTATS Process Flow

PRELIMINARY ASSESSMENT

GATHERING INFORMATION: CODE AND LITERATURE REVIEW

The first stage of updating %BDSSTATS includes reviewing the code and relevant literature. We first reviewed %BDSSTATS code line-by-line looking for obvious opportunities to improve efficiency. For example, we looked for places where IF/ELSE statements could be more nested.

Approach	Description	References
Parallelization		
THREADS	SAS option to enable automatic multithreading of certain compatible processes	1
SAS/CONNECT	Parallelize processing with intentionally-designed subtasks that spawn new SAS sessions	1,4
Reduce I/O		
SAS View/Indexing	Ease repetitive accessing of subsets of datasets	5,7
SASFILES	Hold a dataset in working memory, allowing it to be accessed multiple times without needing to be re-read	7
Code Efficiency		
SQL JOIN / PROC APPEND	Reduce run-time or I/O for individual data merges	5,7

Table 1. Example SAS run-time optimization strategies from literature

We then reviewed the literature on optimizing SAS code runtime. Much of this literature focuses on extremely long runtimes and massive datasets, but our search helped us identify useful approaches, some of which appear in Table 1. Note that this case study does not explore all of these approaches.

INVESTIGATE CURRENT RUNTIME BOTTLENECKS

Next, we analyzed the current %BDSSTATS code to understand the factors contributing to its overall runtime. The current code thus became a "control" we used in experiments described in the next section. I distinguish this from the information gathering step because it requires hands-on investigation of the macro being called from code in an example use case. It involves four steps:

1. Enable macro debug options (e.g. add MPRINT, SYMBOLGEN, etc. to the SAS options statement): this is necessary to fully interpret individual processing steps occurring within the macro.
2. Enable runtime analysis options (e.g. add FULLSTIMER or STIMER to the SAS options statement): this reports run statistics not just for the entire program run but for each individual step within the code.
3. Examine the log and categorize runtime elements.
4. Identify bottlenecks based on individual steps with long runtime or smaller steps that execute multiple times and aggregate to long runtimes.

We could have analyzed the code manually, but that would have been tedious. Instead, we used scripts and SAS code to extract and aggregate the relevant information. Table 2 summarizes some of the largest runtime contributors. Note that we chose a use case without extremely long runtimes to save time conducting these analyses. The code we chose includes five calls to %BDSSTATS to obtain statistics across variable CLASS and BY groups.

Procedure/Step	# Times	Est Total Time (s)	% of Run Time
%MACROUSE inside %MAXLENGTH	337	17.5	24.0
PROC SORT &INDATA	54	9.4	12.9
PROC SQL (nmiss)	135	7	9.6
PROC UNIVARIATE	102	6	8.2
PROC FREQ	50	3	4.1

Table 2. Runtime bottlenecks

As Table 2 shows, nearly a quarter of %BDSSTATS runtime is from a CBAR macro, %MACROUSE. That macro is called inside another CBAR macro, %MAXLENGTH, which is a utility that assists in merging datasets without truncating variables. %MACROUSE helps the macro programming team track how statisticians use its macros. Because %MAXLENGTH is called often in %BDSSTATS, the %MACROUSE call occurs over 300 times, so even a small individual step runtime (often ~0.05s according to the log output) represents a significant portion of the total runtime.

The second-largest runtime contributor is a single PROC SORT step that sorts the dataset fed into the macro (&INDATA) to facilitate BY GROUP processing. The first code review found nothing unusual about this step, but the runtime analysis found that it contributed over 10% of the total program runtime, which caused us to review this step more closely. This helped us realize that it did not need to be inside the main program loop.

The next three steps are required for %BDSSTATS code to function. Redesigning the PROC UNIVARIATE and PROC FREQ steps could make them more efficient and are on the list of options to explore in the next section. The code runs these PROCs once per parameter/variable and outputs all enabled statistics even if the user did not request them. We identified two potential ways to reduce runtime for these processes: Limiting the output to only the requested statistics and redesigning the code to combine all parameters and variables into single PROC calls.

As Table 3 shows, the FULLSTIMER output also showed that multiple steps take 0.00s but still contribute significantly to the overall code runtime. There are 4383 steps that each took 0.00s but contributed ~40% of the overall runtime. It is thus important to explore possible improvements to any repetitive task within a code, even when FULLSTIMER doesn't capture its contribution to the full runtime.

Procedure/Step	# Times	Est Total Time (s)	% of Run Time
Steps shown in Table 2	678	42.9	58.7
Steps taking < 0.00s	4383	21.0	40.3

Table 3. Overall runtime analysis

SUMMARIZE PRELIMINARY OPTIONS

We then summarized the preliminary options for improving code. Figure 3 shows the case-study version of this options matrix. Our initial runtime analysis allowed some of these boxes to be small as we had quantitative data on how much they contributed to our code runtime. Others, identified during our literature review were uncertain, leading to larger boxes. This is completely acceptable at this point.

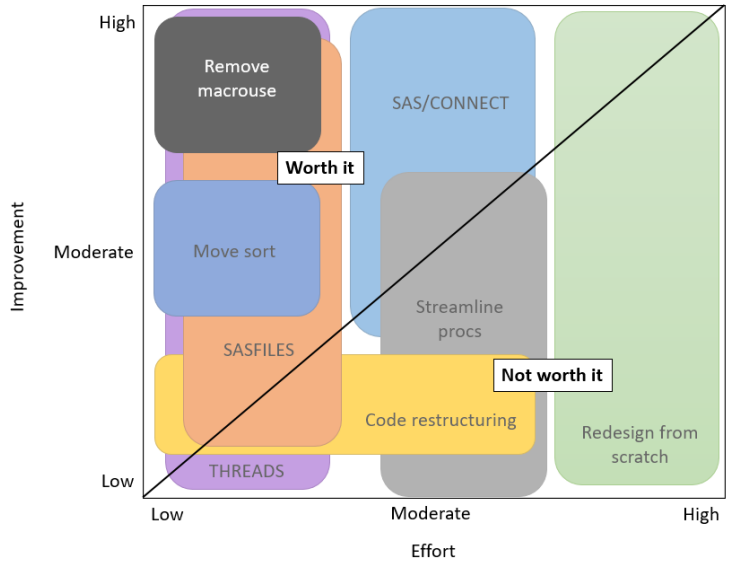

Figure 3. Preliminary options matrix

EXPERIMENTATION

Our next goal was to refine the plot in Figure 3 to determine which changes to implement. Changes in the top left of the plot offer relatively high improvement with relatively low effort and may be worth implementing while those in the lower right of the plot are not.

What about the changes that straggle both sides of the "worth it or not worth it" line? To answer this question, we conducted limited experiments on each approach that straddles the line to roughly estimate its potential impact. The objective of these experiments is to refine the options matrix to the degree required to aid in decision-making, not to completely capture how the changes would be put into practice, so a fully-implementable solution is not required at this stage.

We classify our experiments by their required approach:

1. those that can call the current production version of %BDSSTATS,
2. those that require a modification to a test version of %BDSSTATS, and

3. those that require mocking up only a portion of the %BDSSTATS code.

Table 4 shows each experiment we performed for this case study.

Category	Description	Experiment
1	Can use the production macro	THREADS
		SAS/CONNECT
		SASFILES
2	Requires modifying a test version of the macro	Remove %MACROUSE
		Move PROC SORT
3	Requires mocking up only a portion of test code	Streamline PROCs

Table 4. Categorizing experiment type: (1) use production macro, (2) modify test macro, (3) code mock-up

CATEGORY 1 EXPERIMENT: USING A PRODUCTION MACRO

Testing SAS/CONNECT is an example of a Category 1 experiment. Implementing SAS/CONNECT within %BDSSTATS would:

- focus only on the parts of the code that loop over the requested parameters/variables,
- split the parsed list of parameters/variables into multiple SAS/CONNECT subtasks, and
- recombine the data after each subtask has executed the main program loop.

To run the experiment, we manually parsed the list of variables used in the control code. %BDSSTATS is then called inside each SAS/CONNECT subtask. This experiment did not handle recombination of data, limiting programmer effort, and took only a few minutes to write. But this experiment provided a reasonable estimate of the impact of making this change.

CATEGORY 2 EXPERIMENT: MODIFYING A TEST MACRO

The only way to test removal of %MACROUSE from %MAXLENGTH and the impact of moving the PROC SORT outside of the loop was to actually do it. These are simple changes, so we copied %BDSSTATS and %MAXLENGTH into the testing directory and modified them. Using these modified versions only required two changes to the control experiment: changing the macro call name to the appropriate test macro and changing the %INCLUDE statement.

CATEGORY 3 EXPERIMENT: MOCKING UP CODE

Our final experiment category mocked up a portion of the %BDSSTATS code to test possible changes to the PROCs. We created a separate control code because this mock-up is not directly comparable to the control run of %BDSSTATS. Here, we looped PROC UNIVERATE over all variables required to obtain the requested statistics. We similarly looped over PROC FREQ, matching the same output we were obtaining from the original control code. We revised this new control code to test each proposed modification to the PROCs:

- To reduce statistics: We deleted most statistics from these two looping PROCs.
- To combine PROC calls: We used a single PROC UNIVARIATE and PROC FREQ for all variables (with all statistics from the control code) and deleted the loops.

Notable here, the effects of combining the PROCs could be greater than these experiments measured. %BDSSTATS generates a new dataset for every parameter variable, so this macro loop requires multiple steps to merge the data, steps that likely encompass several of the 0.00s-steps discussed earlier. But we limited the scope of this experiment to reduce programmer effort.

SECONDARY EXPERIMENTS

Running a few experiments highlighted some options as promising (the next section will discuss these results). Secondary experiments identify these candidates and collate them to assess their combined impact. For example, we found that raising the number of SAS/CONNECT subtasks from 3 to 6 yielded significant benefit only when we made no other changes to the code.

ANALYSIS CONSIDERATIONS

The FULLSTIMER option outputs various run statistics that can used to compare experiments. CPU time may be a more accurate way to estimate the runtime of each code because other processes occurring on the system when code is running may affect real time. But we included parallelization approaches (THREADS, SAS/CONNECT) in this case study so CPU time for our experiments would overestimate their runtime. This required us to use real time for comparison. To compensate for the effects of other system processes, we repeated this series of experiments over 150 times.

Figure 4 shows the real times we extracted from each run of the control code. The first ~60 runs occurred during business hours and the remaining 100 occurred after hours. Real time varied significant during business hours and, though not shown in Figure 4, occasionally the control ran faster than some of the experiments run in that same window of time. Real time varied much less after hours, making this the ideal time to conduct experiments and helping ensure they did not slow other critical programs.

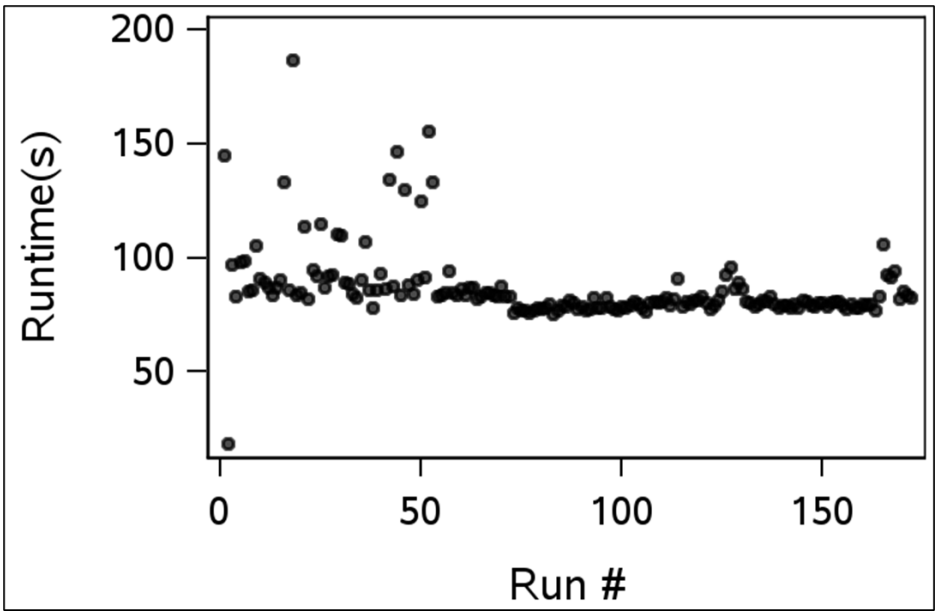

Figure 4. Runtime for each run of the control experiment

CASE STUDY FINDINGS

EXPERIMENTAL RESULTS

Figure 5 shows the averaged runtime for each primary and secondary experiment as the percent reduction in runtime relative to the associated control code. The Category 1 and 2 experiments showed a potential runtime improvement of up to almost 70% compared with the full control code (by removing %MACROUSE, moving the PROC SORT, and implementing SAS/CONNECT). Reducing output statistics in the PROC mock-up experiments did not significant affect runtime; the slight increase in runtime is likely due to random variability. In contrast, combining the PROC calls eliminated nearly 80% of the PROC run time vs. the control.

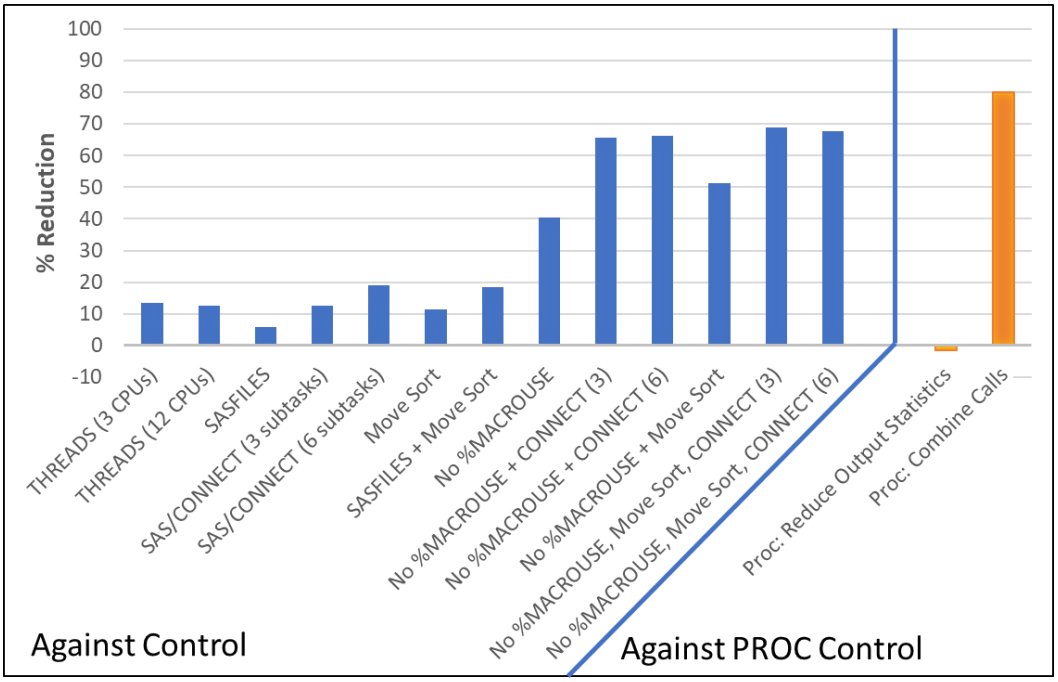

Figure 5. Percent reduction in average real run time (vs. associated control program) for each experiment

SELECTING OPTIONS TO IMPLEMENT

Approach	Level of Effort	Expected Improvement	Decision
Remove %MACROUSE from %MAXLENGTH	Low	High	Implement
Move PROC SORT	Low	High	Implement
Combine PROCs	Moderate	Moderate	Planned future
SAS/CONNECT	Moderate	Moderate	Possible future

Table 5. Refined list of options for runtime improvement

These experiments showed refine the improvement we could expect from implementing each possible change in the options matrix. The experiments thus helped us narrow the list of possible changes to those in Table 5, which shows the decision of the macro programming team to include an option in the current planned update of %BDSSTATS or move it to a future update. We removed many of the candidate options that early testing identified because they fell into the lower-right of the options matrix (not worth it).

For example, we can expect streamlining PROCs by combining PROC calls to yield a fairly large improvement. But this would mean making major updates to the looping part of the code and some of the subsequent processing steps. Our priority in updating %BDSSTATS at this time is to improve its functionality for users (e.g. adding new output statistics). It is more important to release an update quickly, and we opted to only implement the “low effort” changes for now. Streamlining the PROCs may still be pursued in a subsequent update as time and workload permit.

SUMMARY

This paper describes a process for improving macro runtime and how we applied it to optimize the %BDSSTATS macro. It is important to balance the expected runtime improvement with the time programmers will need to spend updating macros, both in terms of hours worked and time to completion.

Identifying changes that can reduce macro runtime and estimating how long it will take programmers to implement them can be straightforward. But estimating how much runtime improvement to expect from a specific approach is more complex. The process this paper describes therefore includes experiments to help refine these estimates and inform the decisions programmers make. Our team, for example, had no experience with SASFILES and found it hard to estimate its impact on %BDSSTATS. Our experiments found that implementing SASFILES was slightly more complex than it first appeared and yielding little improvement to runtime, helping us eliminate it from our options matrix.

An analysis of the runtime of individual processing steps (leveraging STIMER or FULLSTIMER output) can aid in the identification of bottlenecking lines of code. The focus here should be on steps that, individually or in aggregate, contribute a large portion of the overall runtime, but it is also important not to lose sight of the impact of 0.00s steps that can also aggregate through sheer repetition. As an example, the initial assessment of bottlenecks in this case study showed that the %MACROUSE call within %MAXLENGTH contributed roughly 24% of the runtime of the control code. However, it's removal also eliminated some of the 0.00s steps, so the experiment removing %MACROUSE actually reduced runtime by ~40%.

Readers may ask if the benefits of this process justify the time and effort required to implement it. Setting up and conducting the experiments we described took programmer and calendar time that programmers could have spent updating %BDSSTATS. One programmer spent 12 hours over roughly one month, or 10% of their time. The programmer conceived the approach, developed the scripts and companion SAS codes for automating experiments, and analyzed their results. But many of these are reusable for future optimization projects.

We expect the full macro update process will consume 60% of one programmer's time over two months with the options selected, which is largely due to the complexity of folding in the new statistics and not the runtime reduction options. Developing and refining the options matrix represents less than 10% of the entire macro update process. Our experiments show we reduced runtimes over 50%; they also prepared us to implement this process for future macro updates if we decide to pursue greater runtime reductions.

This particular macro was called over 2000 times in the past calendar year. If we judiciously estimate that the average call to %BDSSTATS takes 3 minutes (some are much faster, and some are much longer, depending on the statistics requested), then this 50% reduction in macro runtime represents >50hrs in runtime savings per calendar year, more than 4X the programmer time required to investigate and implement the changes.

Investing time in assessing options for improving runtimes using this process can help to estimate the impact of those options more precisely and determine whether to pursue them during macro updates.

ACKNOWLEDGMENTS

This work was supported by the Statistical and Data Management Center (SDMC), of the Advancing Clinical Therapeutics Globally for HIV/AIDS and Other Infections (ACTG), under the National Institute of Allergy and Infectious Diseases (NIAID) grant No. UM1 AI068634.

This work was supported by the Statistical and Data Management Center (SDMC) of the International Maternal Pediatric Adolescent AIDS Clinical Trials Network (IMPAACT) under the National Institute of Allergy and Infectious Diseases (NIAID) grant No. UM1 AI068616.

REFERENCES

[1]Barbour, J. 2018 "Parallel Processing with Base SAS." *Pharmasug 2018*. 15-2018.

[2]Brown, T. and Crevar, M. 2016. “Best Practices for Configuring Your I/O Subsystems for SAS 9 Applications.” *SAS 2016*. SAS6761-2016.

[3]DiIorio, F. 2008. “Building the Better Macro: Best Practices for the Design of Reliable, Effective Tools.” *NESUG 2008*.

[4]Haigh, D. 2016, October. “Divide and Conquer—Writing Parallel SAS® Code to Speed Up Your SAS Program.” *The 24th Annual Southeast SAS Users Group (SESUG) Conference. SESUG 2016*. PA-265.

[5]Iyengar, K. 2023. “Best Practices for Efficiency and Code Optimization in SAS Programming.” *Pharmasug 2023*. AP-057.

[6]Meyers, J. 2024. “Tips for Completing Macros Prior to Sharing.” *Pharmasug 2024*. AP-175.

[7]Ruegsegger, S. 2011. “Programming Techniques for Optimizing SAS Throughput.” *NESUG 2011*.

CONTACT INFORMATION

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

Valerie Finnemeyer
Harvard TH Chan School of Public Health
vfinneme@sdac.harvard.edu