Paper SAS4104-2020

What's Bugging You? Find Out with the Interactive SAS® Code Debugger

Jenna Austin, Toshiba Burns-Johnson, Aaron Mays, and Mike Whitcher

SAS Institute Inc.

ABSTRACT

Life's too short to put up with buggy code. Join us as we show you how to use the new SAS® Code Debugger to debug SAS® code in the SAS Function Compiler (FCMP) procedure and in the SAS® Cloud Analytic Services FCMP action set.

SAS Code Debugger is a full-featured, interactive debugger that runs on SAS® Viya® 3.5 and SAS® 9.4M6. The debugger is available as a stand-alone tool, and it is also built into SAS products such as SAS® Model Implementation Platform.

With SAS Code Debugger, you can discover programming problems, improve code quality, and debug complex models with ease. You can also set breakpoints, watch variables, and step into nested functions and subroutines.

This paper introduces SAS Code Debugger, shows you tips for debugging SAS code, and gets you up and running quickly.

INTRODUCTION

Debugging is the process of identifying and removing logic errors from a program. Unlike syntax errors, logic errors do not stop a program from running. Instead, logic errors cause the program to produce unexpected results.

To debug SAS® code, typically, you insert PUT statements at selected places in the code, submit the code, and then examine the values that are displayed in the SAS log. This debugging process might require you to spend considerable time locating the problem.

SAS® Code Debugger is an interactive, web-based solution that helps you streamline the debugging process. With SAS Code Debugger, you can perform the following tasks:

- monitor the execution of SAS code that is submitted through the SAS® Cloud Analytic Services (CAS) FCMP action set in SAS® Viya® 3.5 and through the following procedures in select risk solutions that run on SAS® 9.4M6: PROC FCMP, PROC COMPILE, and PROC HPRISK
- start, pause, and resume program execution
- step into, step over, and step out of functions, subroutines, and methods
- set breakpoints that interrupt an executing program
- monitor and change variable values

In addition to identifying logic errors, you can use SAS Code Debugger to learn how a program works. For example, when you are exploring a new codebase, you can step through the code one line at a time, watch the code flow, and observe the state of the variables. This process provides you with a high-level overview of one path through the code.

INVOKING SAS CODE DEBUGGER

SAS introduced a stand-alone version of SAS Code Debugger with select risk solutions that run on SAS 9.4M6 and with SAS Viya 3.5. SAS also introduced an embedded version of the debugger with SAS® Model Implementation Platform 3.2.

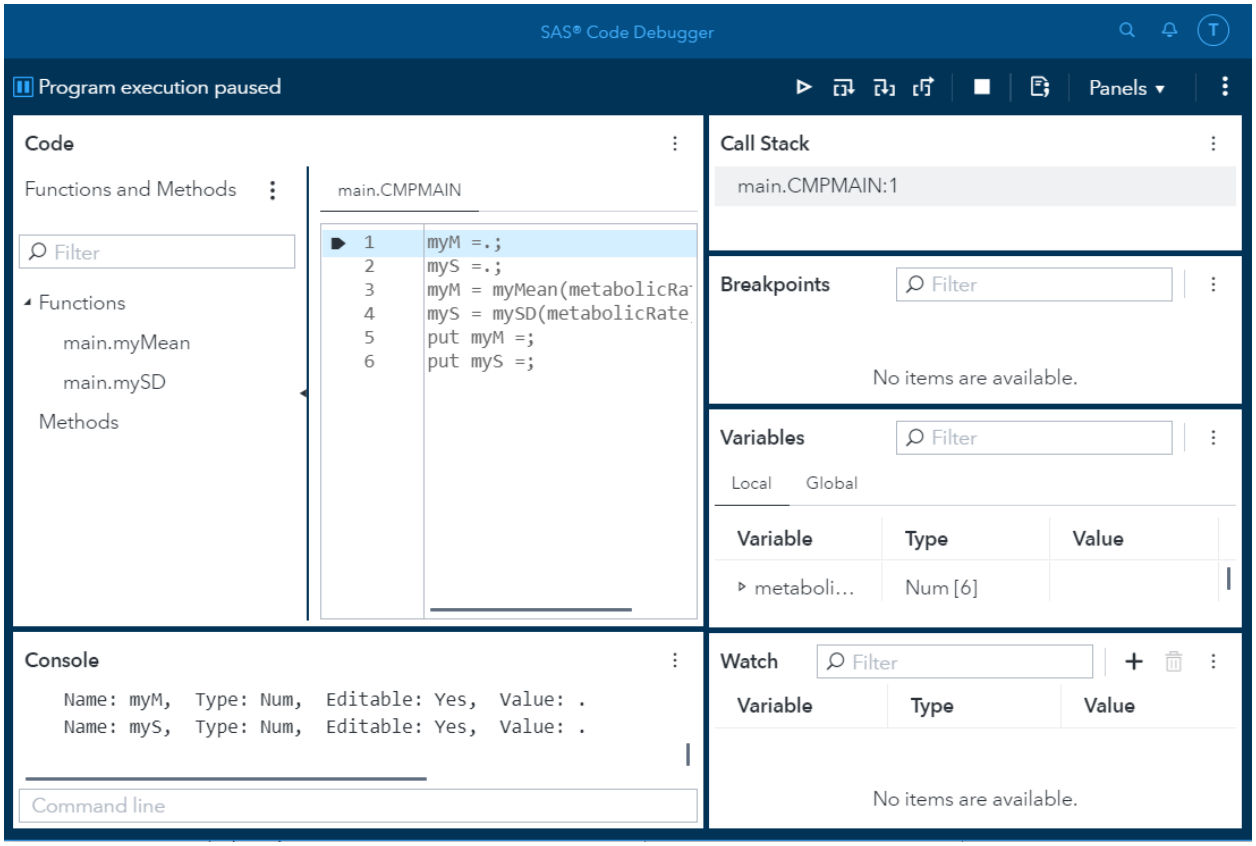
INVOKE THE STAND-ALONE DEBUGGER

To invoke the stand-alone version of SAS Code Debugger, complete the following steps:

1. Enter the URL for SAS Code Debugger in a web browser (for example, http://host.example.com/SASCodeDebugger) and sign in.
2. In the Code panel, copy the OPTIONS statement and add it to the beginning of the code that you want to debug. The OPTIONS statement connects your code to the debugger.
3. To debug code that is submitted through the FCMP action set, add the following DEBUGOPT session option to your CAS statement:

   ```
   cas <session_name> sessopts=(DEBUGOPT="%sysfunc(getoption(debugopt))");
   ```

   This DEBUGOPT session option sets the debug options for the CAS session to the same values that are specified for the SAS session.
4. In a SAS application (for example, SAS® Studio), execute your code. If your code compiles successfully, the code is loaded into the Code panel in SAS Code Debugger and execution is suspended at the first executable line of code.



Display 1: Stand-Alone Version of SAS Code Debugger

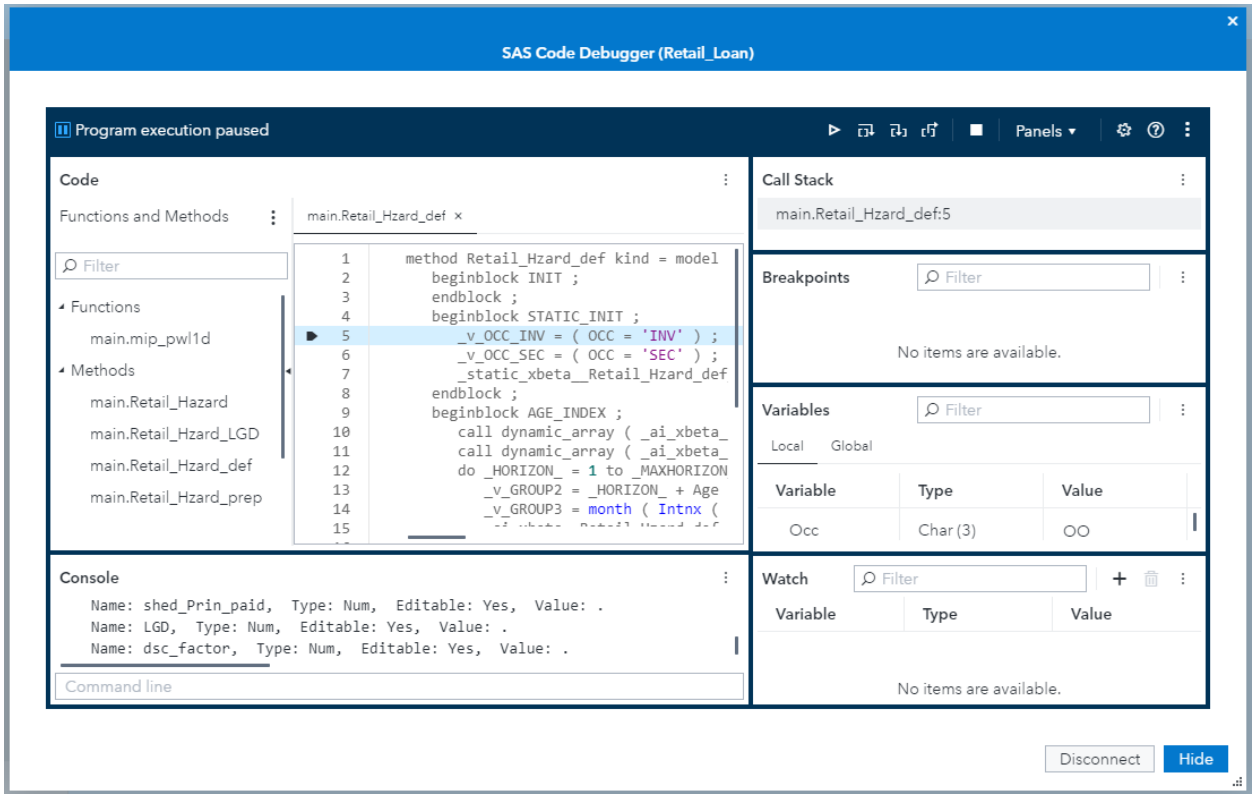
INVOKE THE EMBEDDED DEBUGGER

To invoke the embedded version of SAS Code Debugger, complete the following steps:

1. Enter the URL for SAS Model Implementation Platform in a web browser (for example, http://host.example.com/SASModelImplementationPlatform) and sign in.
2. Create or edit an analysis run or a model unit test, and select Use SAS Code Debugger in the Debugger/Trace field.

   In order to use SAS Code Debugger to debug your analysis runs, distribute your portfolio data on one node and one thread only.

3. Submit the analysis run or the model unit test. SAS Model Implementation Platform loads the generated code into the Code panel in SAS Code Debugger and suspends execution at the first executable line of code.



Display 2: Embedded Version of SAS Code Debugger

EXPLORING THE USER INTERFACE

SAS Code Debugger provides an interactive debugging environment that consists of a toolbar and six panels: Code, Console, Call Stack, Breakpoints, Variables, and Watch.

To create a custom user experience, you can rearrange and resize the panels and you can hide or minimize infrequently used panels.

DEBUGGER TOOLBAR

The debugger toolbar enables you to perform the actions that are listed in Table 1.

Table 1: Actions in the Debugger Toolbar

Action	Button	Keyboard Shortcut
Start or resume program execution	▷	F8
Pause program execution	⏸	F8
Step over a function, subroutine, or method		F9
Step into a function, subroutine, or method		F10
Step out of a function, subroutine, or method		Ctrl + F10
Stop debugging	■	Not applicable
Display the OPTIONS statement (stand-alone version only)		Not applicable
Hide or show panels	Panels	Not applicable
Set the maximum number of array values to display (embedded version only)		Not applicable
Export or import breakpoints and watch variables	⋮	Not applicable

CODE PANEL

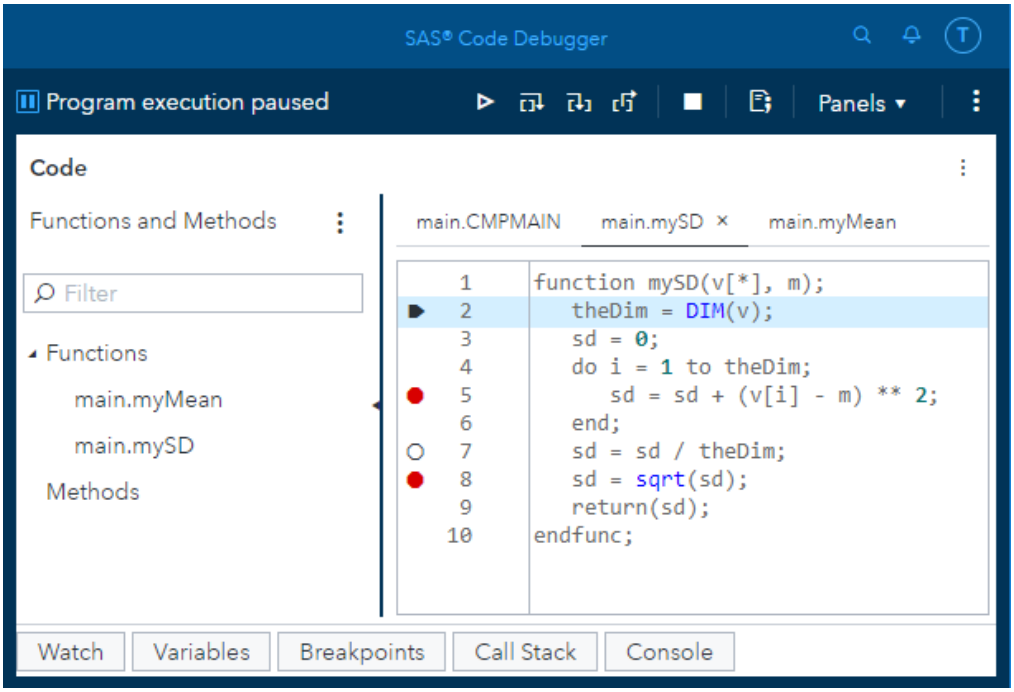
The Code panel provides the following features to assist you with debugging:

- The Functions and Methods section lists the functions, subroutines, and methods that are used in your code and that exist in the same library. You can right-click a function, subroutine, or method to open it in a new tab or to add a breakpoint to the first executable line of code in that function or subroutine, or in each block of that method.
- The tabs section contains a tab for the main code and for each function, subroutine or method that you open. You can right-click the left margin in a tab to add, enable, disable, or remove breakpoints for the corresponding line of code.

  You can add breakpoints only on executable lines of code. Format statements, array definitions, empty lines, and comments, for example, are not executable lines of code.

  Each tab can display the following icons:

  - The ▶ icon identifies the line where the debugger suspended program execution.
  - The ● icon indicates that you enabled a breakpoint for that line of code.
  - The ○ icon indicates that you disabled the breakpoint for that line of code.

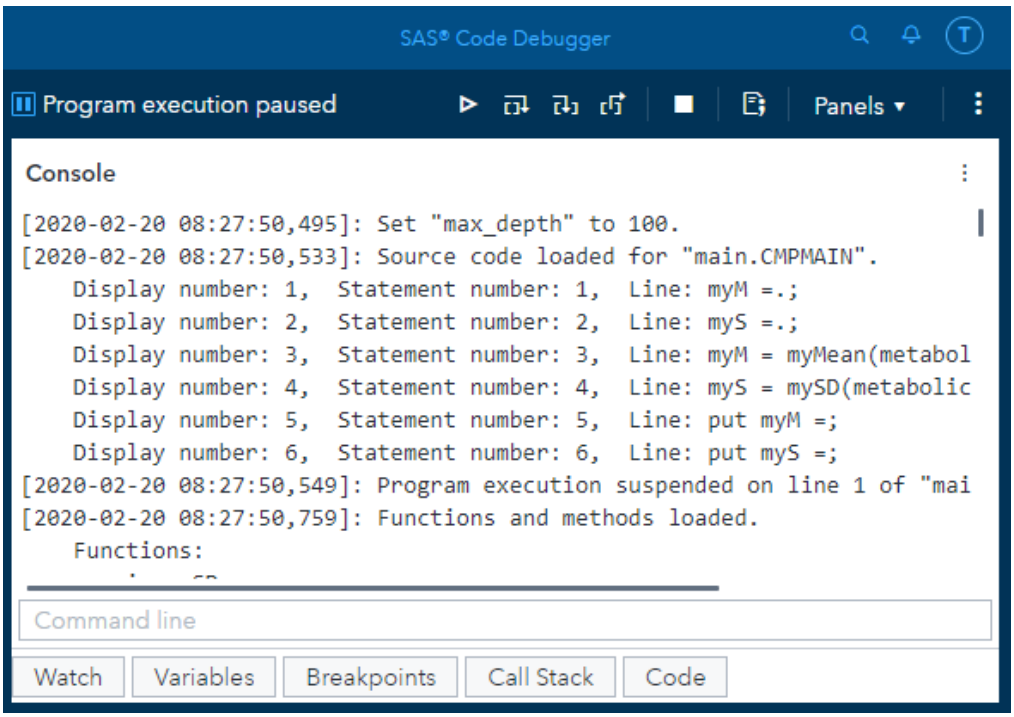


Display 3: Code Panel in SAS Code Debugger

CONSOLE PANEL

The Console panel provides a command line and a running log for your debugger session.

To scroll through the commands that you previously entered, use the up and down arrow keys.



Display 4: Console Panel in SAS Code Debugger

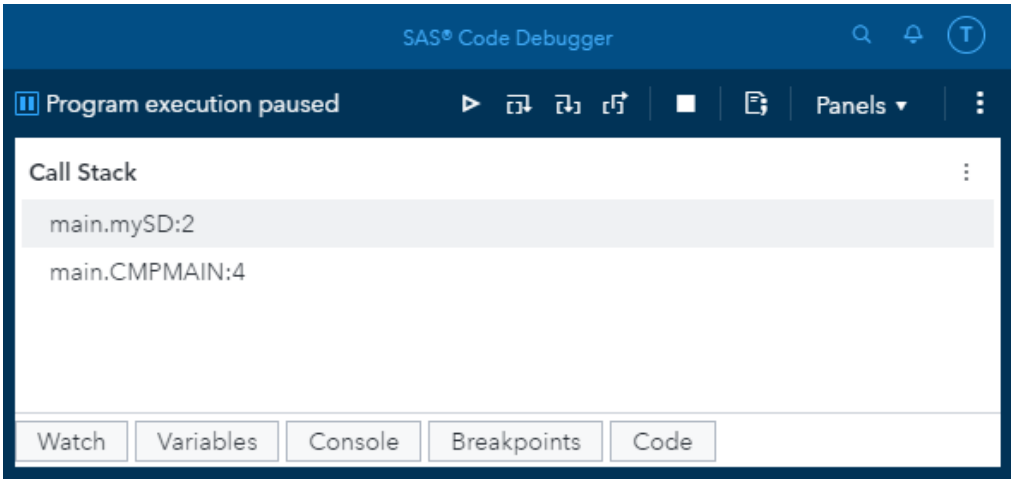
The following commands are the most frequently used:

- break -n *statement_number* | -f *function_method* <-p *package_name*> <-b>
  Adds breakpoints.
- breakclear -a | -l *break_ID*
  Removes breakpoints, where the letter in the *-l break _ID* argument is a lowercase L.
- breakdisable -a | -l *break_ID*
  Disables breakpoints.
- breakenable -a | -l *break_ID*
  Enables breakpoints.
- breaklist
  Lists all the breakpoints.
- breakroutines <-f | -m>
  Adds a breakpoint to the first executable line of code in your functions or subroutines or to the first executable line of code in each block of your methods.
- clear
  Clears the running log.
- go <-l *statement_number* | -o | -n *number_of_statements*>
  Starts or resumes program execution. If the debugger does not encounter any breakpoints, the program runs to completion.
- pause
  Pauses program execution.
- quit
  Terminates the debugger session. The program runs to completion.
- step <-t>
  Executes the current line of code. You can also press the Enter key to issue the STEP command.

For a complete list of commands, see *SAS Code Debugger: User's Guide*.

CALL STACK PANEL

The Call Stack panel lists each stack frame in descending order according to the order in which the program called each function, subroutine, or method. You can click a stack frame to display the code and the variables for the corresponding function, subroutine, or method.



Display 5: Call Stack Panel with Two Stack Frames

The call stack depicted above indicates that line four in the CMPMAIN code called a function named MYSD. The debugger stepped into the MYSD function and paused execution at line two. When the debugger finishes executing the MYSD function, the debugger resumes execution at line five of the CMPMAIN code.

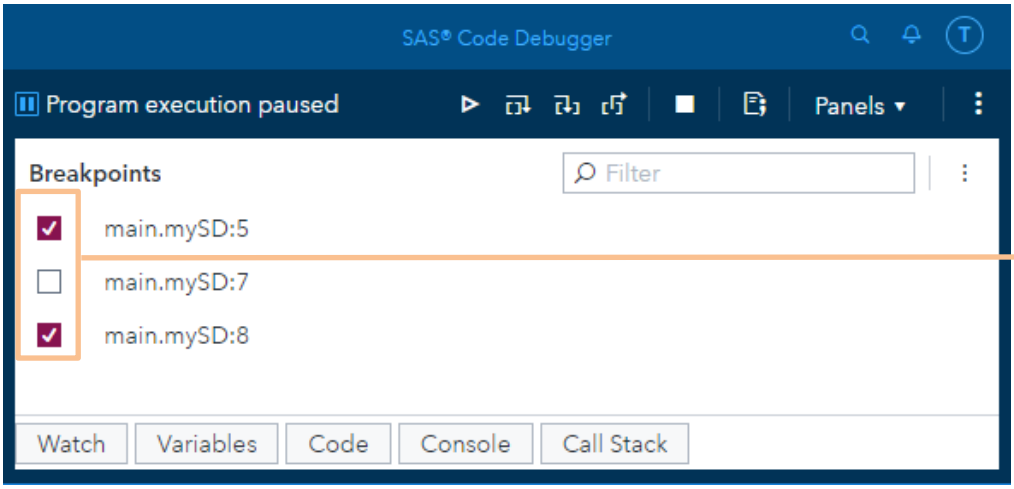
You can step into only user-defined functions, subroutines, and methods. The debugger always steps over encrypted (with PROC FCMP ENCRYPT) and SAS built-in functions and subroutines.

BREAKPOINTS PANEL

A breakpoint identifies a point in the code at which the debugger stops executing the program so that you can perform the following actions:

- examine the current variables
- change the value of a variable
- add watch variables
- issue commands
- review the call stack

The Breakpoints panel lists all the breakpoints that exist in the current debugger session. You can use the Breakpoints panel to enable, disable, or remove breakpoints.



Display 6: Breakpoints Panel with Three Breakpoints

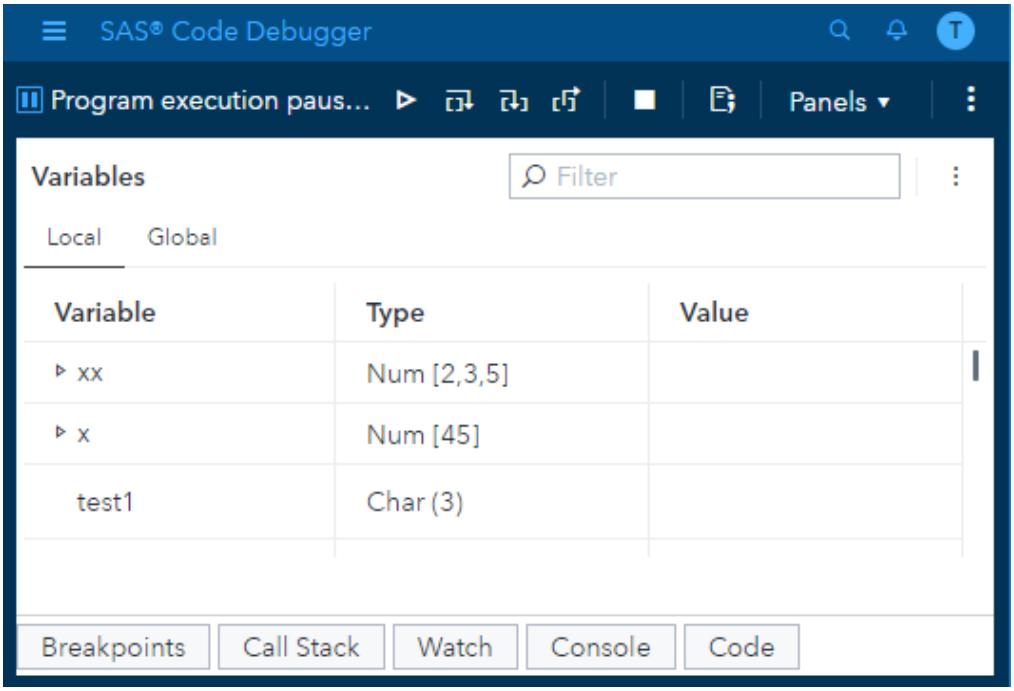
If the debugger does not stop at a breakpoint, this typically means that the input observations did not cause the program to take the corresponding code path. In this case, you can add the breakpoint to a more frequently executed line of code. Then, step toward the code of interest to see what happens.

If the debugger is running in a multithreaded environment and the debugger skips a breakpoint, you can re-run your program as single threaded. This way, the debugger includes every input observation in the debugging process.

VARIABLES PANEL

Variables are an essential part of the debugging process. You can use variables to identify data and logic errors and to pinpoint the lines of code that might be producing unexpected results.

The Variables panel lists all the variables that exist in the program data vector (PDV). As you step through your code, the debugger automatically updates the variables and the variable values.



Display 7: Variables Panel with Three Variables

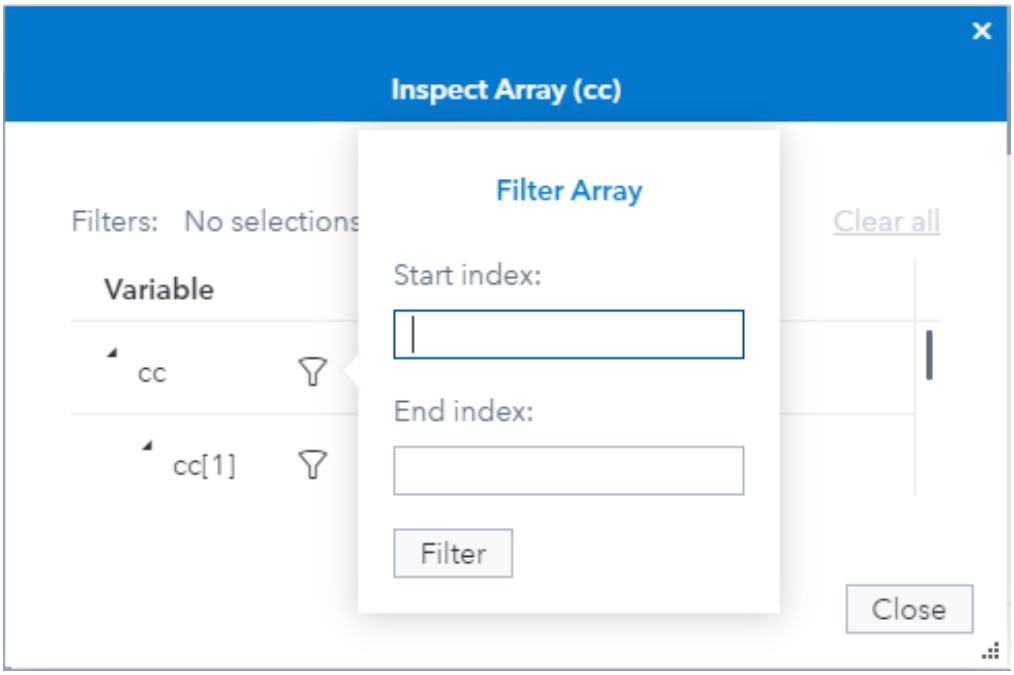
SAS Code Debugger supports variables with the numeric data type (Num) and the fixed-character data type (Char). The debugger also supports arrays of those types.

In the Type column, the debugger uses brackets to designate array variables and parentheses to designate the maximum length of character variables. For example, the type Num [2,3,5] denotes a three-dimensional numeric array variable that has two elements in its first dimension, three elements in its second dimension, and five elements in its third dimension. The type Char (3) denotes a character variable that can contain a maximum of three characters.

In the Variables panel, you can right-click a variable and select Add to watch list to add it to the Watch panel. You can also click a variable's value to manipulate the value in real time.

Typically, you would change a variable's value to investigate the impact of a potential change or to advance your program beyond a previously discovered error.

For an array variable, you can right-click the variable and select Inspect array to examine the array in a separate window. In the Inspect Array window, you can filter the array values by index range so that you can focus on a specific set of values.



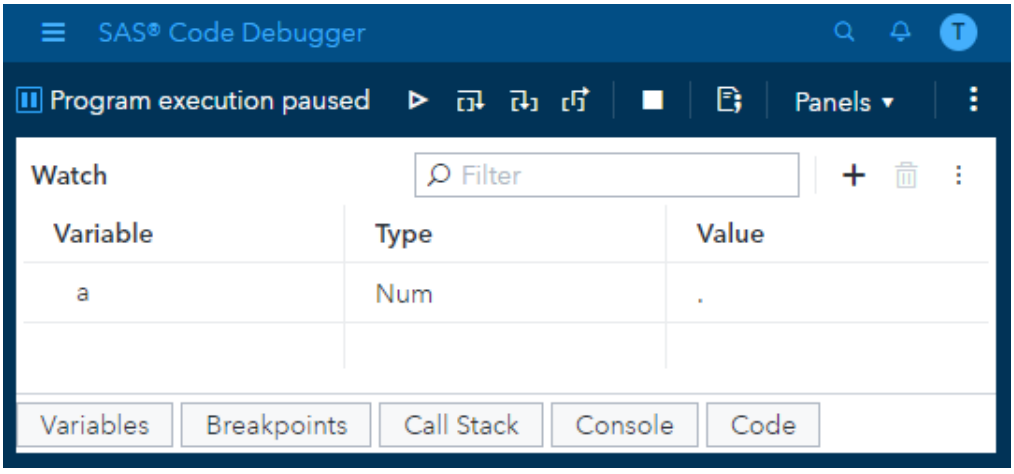
Display 8: Filter Array Pop-Up in the Inspect Array Window

By default, the debugger can display a maximum of 100 values for each array. If an array contains more than 100 values, specify an index range that enables you to inspect the hidden values. For example, you can specify 200 as the start index and 300 as the end index.

To change the maximum number of array values to display, click User name > Settings > SAS Code Debugger > Preferences in the stand-alone version of the debugger, or click ⚙ in the embedded version.

WATCH PANEL

The Watch panel provides a quick and easy way for you to monitor the values of the variables in which you are interested.



Display 9: Watch Panel with One Watch Variable

USING SAS CODE DEBUGGER

EXAMPLE 1: IDENTIFYING A LOGIC ERROR

The following program calculates the mean and the standard deviation for a set of metabolic rates for birds:

```
ods listing file = _dataout;
proc fcmp;
function myMean(v[*]);
   theDim = DIM(v);
   if (theDim < 1) then
      return(.);
   s = 0;
   do i = 1 to theDim;
      s = s + v[i];
   end;
   theMean = s/theDim;
   return(theMean);
endfunc;
function mySD(v[*], m);
   theDim = DIM(v);
   sd = 0;
   do i = 1 to theDim;
      sd = sd + (v[i] - m) ** 2;
   end;
   sd = sd / theDim;
   sd = sqrt(sd);
   return(sd);
endfunc;
array metabolicRate[6] (721.2, 1008.5, 1095.0, 1372.3, 1490.5, 1926.8);
myM =.;
myS =.;
myM = myMean(metabolicRate);
myS = mySD(metabolicRate, myM);
put myM =;
put myS =;
quit;
%let _DATAOUT_NAME = results.txt;
ods listing close;
```

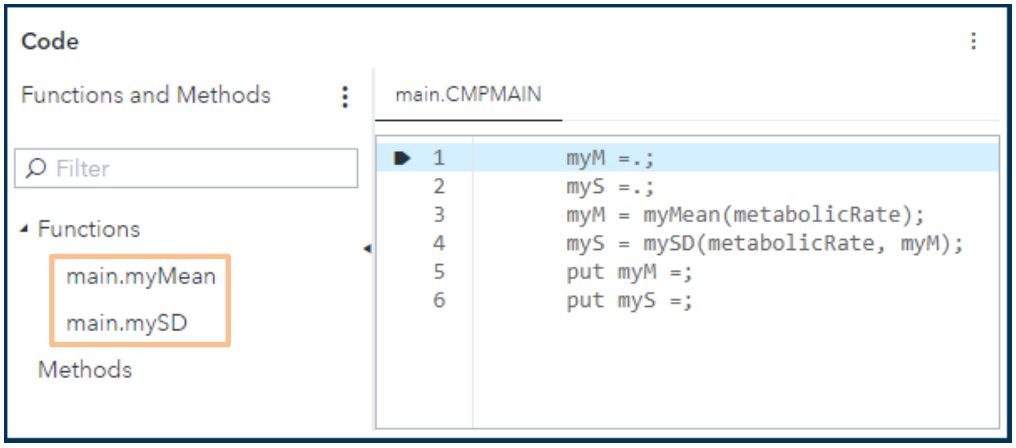
A manual calculation of the mean and the standard deviation shows that the program incorrectly calculates the standard deviation.

Table 2: Program Output Versus Manually Calculated Output

Statistic	Program Output	Manually Calculated Output
Mean (myM)	1269.05	1269.05
Standard deviation (myS)	385.28237588	422.05569656

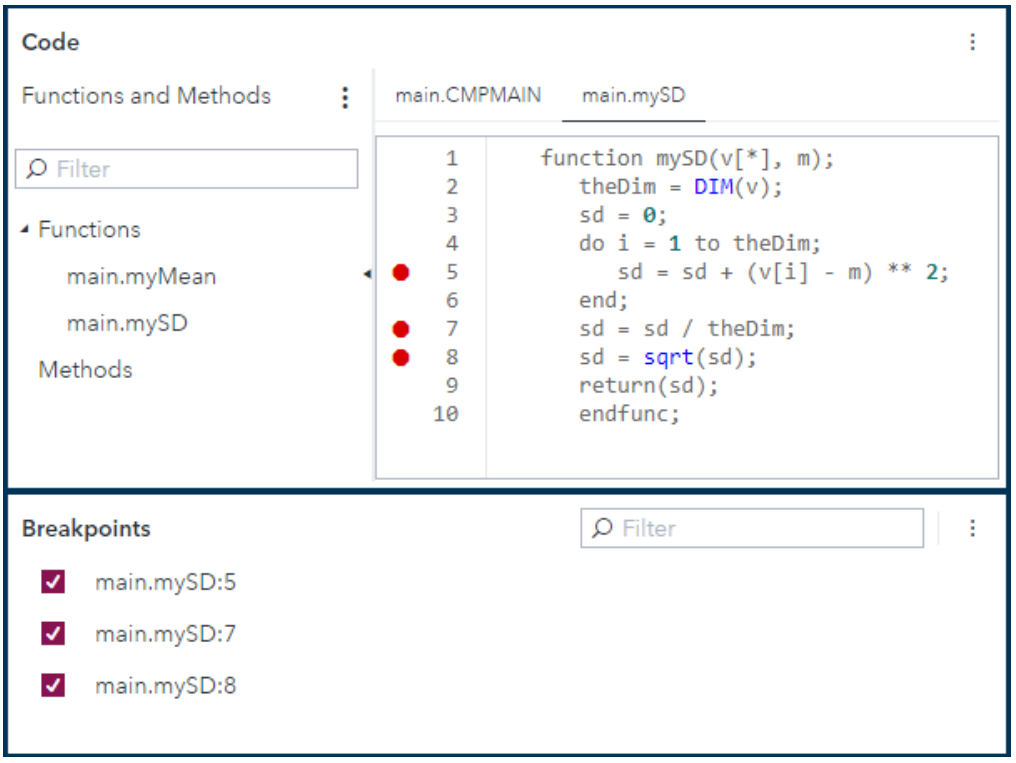
To debug the program, complete the following steps:

1. Invoke the stand-alone debugger. The program contains two functions: MYMEAN (calculates the mean) and MYSD (calculates the standard deviation).



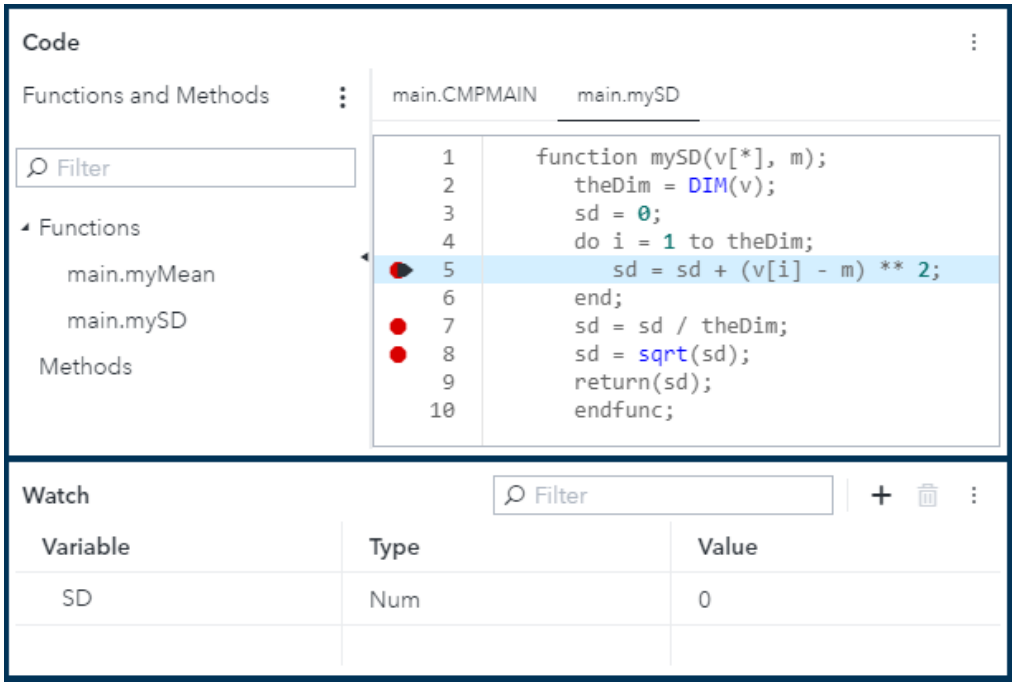
Display 10: Functions in the Sample Program

2. Set breakpoints in the MYSD function.
   a. Double-click main.mySD to open the function.
   b. Set a breakpoint on each line that calculates a portion of the standard deviation (SD variable). Click the margin that precedes each of the lines 5, 7, and 8. The ● icon appears before each line number, and each breakpoint appears in the Breakpoints panel.



Display 11: Breakpoints Added to the Sample Program

3. Make the SD variable a watch variable so that you can monitor its value as you step through the program.
   a. Click + in the Watch panel.
   b. Type SD in the Variable name field.
   c. Click Save. The variable name is grayed out because the variable is not in scope at the current point in the program.
4. Click ▷ in the debugger toolbar to start executing the program. The debugger pauses the program at the first breakpoint. The SD variable is now in scope and is set to 0.



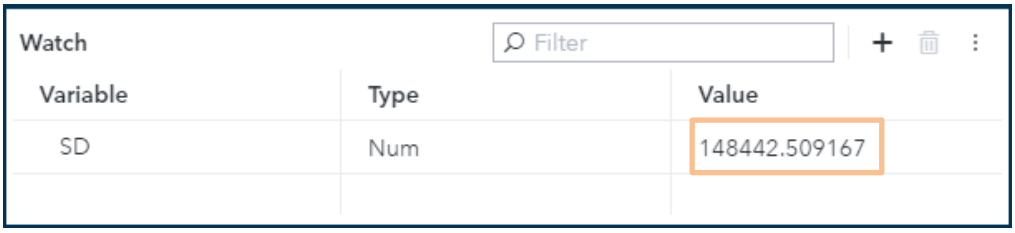
Display 12: First Breakpoint in the Sample Program

5. Click ⤵ in the debugger toolbar six times to step through the DO loop. The program output matches the manually calculated output for each step.

Table 3: Values of the SD Variable for the DO Loop

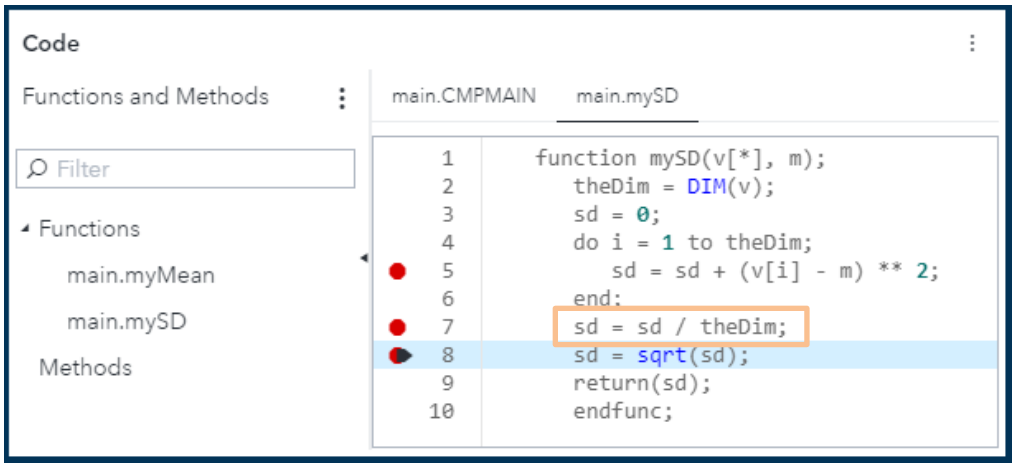
Step	Program Output	Manually Calculated Output
1	300139.6225	300139.6225
2	368025.925	368025.925
3	398319.3275	398319.3275
4	408979.89	408979.89
5	458019.9925	458019.9925
6	890655.055	890655.055

6. Click ⤵ to debug the line of code that is at the next breakpoint (line 7). The program output of 148442.509167 does not match the manually calculated output of 178131.011.



Display 13: Current Value of the SD Watch Variable

The code on line 7 is incorrect. A manual calculation of the standard deviation shows that the denominator should be `(theDim – 1)`.

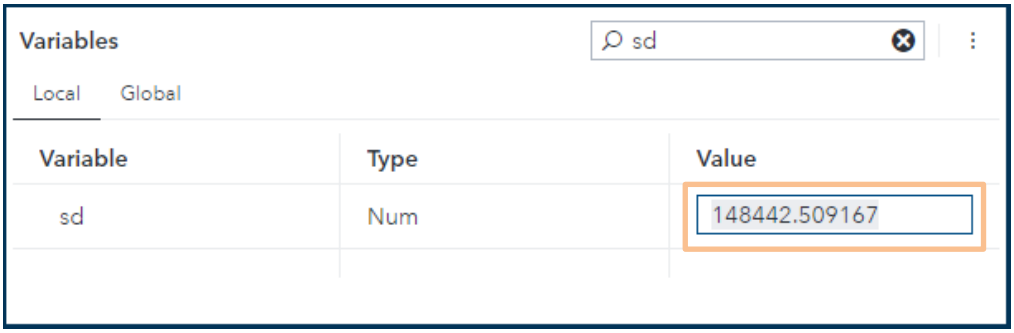


Display 14: Line 7 in the MYSD function

7. Change the value of the SD variable to 178131.011 so that you can move beyond the error and continue debugging.

   a. In the Variables panel, type SD in the Filter field.

   b. Click the value of the SD variable. An editable field appears.



Display 15: Editable Field in the Value Column

   c. Change the value to 178131.011.

   d. Press Enter to save the updated value.

8. Click ⤵ to debug the line of code that is at the next breakpoint (line 8). The SD variable has the expected value of 422.055696561, so you've identified all the errors.

9. In the Console panel, type quit in the Command line field to stop debugging the code.

10. Update the program as follows and re-run it:

```
options CMPOPT = NONE;
ods listing file = _dataout;
proc fcmp;
function myMean(v[*]);
   theDim = DIM(v);
   if (theDim < 1) then
      return(.);
   s = 0;
   do i = 1 to theDim;
      s = s + v[i];
   end;
   theMean = s/theDim;
   return(theMean);
endfunc;
function mySD(v[*], m);
   theDim = DIM(v);
   sd = 0;
   do i = 1 to theDim;
      sd = sd + (v[i] - m) ** 2;
   end;
   sd = sd / (theDim - 1);
   sd = sqrt(sd);
   return(sd);
endfunc;
array metabolicRate[6] (721.2, 1008.5, 1095.0, 1372.3, 1490.5, 1926.8);
myM =.;
myS =.;
myM = myMean(metabolicRate);
myS = mySD(metabolicRate, myM);
put myM =;
put myS =;
quit;
%let _DATAOUT_NAME = results.txt;
ods listing close;
```

For SAS 9, this OPTIONS statement disconnects the program from SAS Code Debugger and removes any other compiler optimizations.

For SAS Viya, specify the following statement: options NODEBUGOPT;

The standard deviation is 422.05569656. You have successfully debugged the program.

EXAMPLE 2: EXPLORING NEW CODE

Let's assume that you inherited a SAS program that implements a computational model. Typically, the program reads an input table into the program space, evaluates the model variables as the model code executes, and generates some type of probability of occurrence or likelihood estimate. Because computational models are often proprietary, let's use a guessing game to demonstrate how you can use SAS Viya and SAS Code Debugger to explore these powerful models.

Consider the following program, which simulates the binary search algorithm to guess a random number between 1 and 100,000:

```
/*<insert_OPTIONS_statement>*/
cas mysession sessopts=(DEBUGOPT="%sysfunc(getoption(debugopt))");
libname sascas1 cas sessref="mysession" caslib="CASUSER";
data sascas1.hlin;
  keep answer;
  call streaminit(int(time()));
  do i=1 to 1000000;
     answer = int(rand('uniform') * 100000);
     output;
  end;
run;
proc cas;
  source highlowsrc;
    function checkguess(highguess, lowguess, currentguess, answer);
      outargs highguess, lowguess;
      if currentguess > answer then do;
        if currentguess < highguess then highguess = currentguess;
        return(1);
      end;
      else if currentguess < answer then do;
        if currentguess > lowguess then lowguess = currentguess;
        return(-1);
      end;
      highguess = currentguess;
      lowguess = currentguess;
      return(0);
    endfunc;
    function highlow(answer);
      call streaminit(_RANKID_ * _THREADID_ * int(time()) + 1);
      numguesses = 0;
      currentguess = int(rand('uniform') * 100000);
      lowguess = 0;
      highguess = 100000;
      hl = 1;
      do until (hl = 0 or (numguesses > 40));
        hl = checkguess(highguess, lowguess, currentguess, answer);
        numguesses = numguesses + 1;
        if hl > 0 then do;
          nextrange = int((currentguess - lowguess)/2);
          if nextrange < 1 then nextrange = 1;
             currentguess = currentguess - nextrange;
        end;
        else if (hl < 0) then do;
          nextrange = int((highguess - currentguess)/2);
          if nextrange < 1 then nextrange = 1;
             currentguess = currentguess + nextrange;
         end;
```

```
      end;
      return(numguesses);
    endfunc;
    keep threadid numguesses answer;
    threadid = _THREADID_;
    numguesses = highlow(answer);
  endsource;
  action fcmpact.runProgram /
    inputData = {caslib="CASUSER" name="hlin"}
    outputData = {caslib="CASUSER" name="highlowResults" replace=1}
    routineCode = highlowsrc;
run;
quit;
proc freq data=sascas1.highlowresults;
  table threadid*numguesses/ nocol norow nocum nopercent;
run;
```

To learn how the program works, complete the following steps:

1. Invoke the stand-alone debugger. The program contains two functions: CHECKGUESS and HIGHLOW.
2. Open each function and review the code. Notice the following:
   - For the DATA step:
     - The debugger cannot access the DATA step.
     - The program uses the RAND function inside the DATA step to generate one million random numbers in the range 1−100,000.
     - The program uses the CAS LIBNAME engine to automatically partition and distribute the generated numbers across all the nodes in the CAS server as CAS writes the numbers to the output table.
   - For the PROC CAS SOURCE statement:
     - The PROC CAS SOURCE statement specifies the FCMP source code that the program uses to play the guessing game, and it assigns that code to an object named HIGHLOWSRC. The SOURCE statement enables you to embed your FCMP code directly instead of as a text string so that you can benefit from syntax highlighting and minimize issues with quotation marks.
     - The CHECKGUESS function determines whether the current guess is too high, too low, or correct.
     - The HIGHLOW function adjusts the range of numbers from which to guess and calculates the next guess.
     - The KEEP statement specifies which variables will be stored in the CAS output table.
   - For the FCMPACT.RUNPROGRAM action:
     - The inputData option specifies the input CAS table, which contains the random numbers to be guessed.

    - The routineCode option executes the code that is included in the HIGHLOWSRC object on each available thread in parallel. This divide and conquer approach can mean dramatic performance improvements for complex programs and large data tables.
    - The outputData option accumulates the results for all the threads in a single CAS table.
  - For the FREQ procedure:
    - The FREQ procedure is CAS enabled, which means that this procedure works with both SAS 9 and CAS engines.
    - The FREQ procedure uses the location of the input data to determine which SAS Access engine to use.
3. Click ⮓ to step through the code. Notice the following:
   - The debugger steps into the HIGHLOW function, but it steps over the STREAMINIT function. The STREAMINIT function is a SAS function that generates the seed value for subsequent pseudo-random number generation by the RAND function. The debugger can step into only user-defined functions.
   - The value of the _THREADID_ variable does not change. SAS Code Debugger debugs on only one node and one thread in a multithreaded environment. Typically, the debugger connects to the lowest numbered thread on the lowest ranked node in that environment.
   - The program uses the halving approach to hone in on the answer. To see the halving approach in action, watch the LOWGUESS, HIGHGUESS, CURRENTGUESS, and ANSWER variables.
   - When the thread guesses an answer correctly, the debugger returns control to the CAS server. The CAS server writes the results to the output table, queues the next random number to be guessed, and returns control to the debugger. This sequence repeats for each observation in the input data.
4. Press ■ to run the program to completion.
5. View the output of the FREQ procedure. The following sample output reveals the following information:
   - The input data was distributed across 24 threads. Threads 1−7 processed 65,536 observations. Threads 8−23 processed 32,768 observations, and thread 24 processed 16,960 observations.
   - Although you debugged on only one thread, in this case thread 1, the CAS server executed the code in parallel on the other 23 threads.
   - Each thread took at most 19 attempts to guess the correct answer.

   Note: Your results might differ from the sample output because the STREAMINIT function generates a different seed each time you execute the program.

The FREQ Procedure

Frequency

Table of threadid by numguesses

threadid	numguesses 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	Total
1	0	1	0	4	14	22	46	101	168	350	683	1328	2543	5012	9417	16144	21902	7801	0	65536
2	0	3	1	5	12	23	49	95	164	335	703	1314	2585	5121	9426	16135	21836	7729	0	65536
3	0	3	7	3	7	22	32	73	169	315	659	1232	2574	4975	9406	16204	21990	7864	1	65536
4	0	1	0	5	6	20	47	88	167	351	641	1293	2623	5239	9451	15989	21901	7714	0	65536
5	0	1	2	4	14	15	46	83	177	335	626	1301	2579	5043	9477	16151	21878	7804	0	65536
6	0	0	3	2	9	17	46	93	201	322	647	1350	2530	5066	9400	16179	21977	7694	0	65536
7	2	3	2	7	10	26	36	95	186	323	645	1288	2514	5006	9531	16260	21850	7751	1	65536
8	1	2	3	1	2	14	25	40	83	178	326	650	1277	2548	4725	8118	10934	3840	1	32768
9	0	0	1	3	8	8	19	45	85	192	327	674	1298	2474	4654	8131	10958	3891	0	32768
10	1	2	0	0	7	8	16	55	79	175	328	692	1335	2523	4745	7966	10892	3944	0	32768
11	0	1	0	5	5	9	19	37	70	138	345	696	1341	2585	4643	8012	11015	3847	0	32768
12	1	1	2	1	5	5	22	48	89	161	338	650	1289	2550	4796	8190	10853	3767	0	32768
13	0	0	0	5	9	13	19	38	87	181	364	647	1296	2452	4865	8072	10872	3847	1	32768
14	0	0	0	3	3	2	22	33	75	153	336	640	1270	2472	4822	8164	10839	3934	0	32768
15	0	0	2	2	7	9	20	39	97	164	306	678	1298	2529	4638	8049	10985	3945	0	32768
16	0	0	0	5	6	13	15	40	78	158	303	736	1330	2484	4673	8118	10836	3973	0	32768
17	0	0	1	5	5	9	24	34	89	173	330	642	1319	2557	4705	8166	10852	3857	0	32768
18	0	0	3	2	4	8	24	44	87	165	342	677	1247	2564	4663	8154	10917	3867	0	32768
19	1	1	2	1	8	9	16	43	71	169	334	610	1265	2586	4813	8041	10956	3842	0	32768
20	0	0	1	2	5	12	26	35	81	173	368	666	1313	2536	4661	7989	11012	3888	0	32768
21	0	2	5	4	5	9	27	39	77	168	347	684	1310	2589	4692	8052	10907	3851	0	32768
22	0	2	2	4	4	13	24	43	99	166	348	616	1256	2486	4686	8025	11068	3926	0	32768
23	1	0	0	1	5	10	21	48	85	159	285	666	1327	2561	4656	8110	11008	3825	0	32768
24	0	1	1	0	4	5	7	22	41	91	190	348	646	1248	2485	4164	5743	1964	0	16960
Total	7	24	38	74	164	301	648	1311	2605	5095	10121	20078	39365	77206	144030	246583	333981	118365	4	1000000

Display 16: Sample Output for the FREQ Procedure

CONCLUSION

Whether you are debugging your FCMP code or exploring new FCMP code, SAS Code Debugger has what you need to quickly visualize how your program runs and to identify any issues. With its easy-to-use, interactive interface, SAS Code Debugger is sure to please both new and experienced SAS programmers.

The authors of this paper have enjoyed introducing SAS Code Debugger to the SAS user community. We sincerely hope that you will give it a try.

REFERENCES

SAS Institute Inc. 2019. *SAS Code Debugger: User's Guide*. Cary, NC: SAS Institute Inc.

SAS Institute Inc. 2019. *SAS Model Implementation Platform: User's Guide*. Cary, NC: SAS Institute Inc.

ACKNOWLEDGMENTS

The authors would like to offer a special thanks to Chris Johns of SAS Institute for providing your technical expertise during the creation of this paper.

RECOMMENDED READING

- *SAS Cloud Analytic Services: User's Guide*
- *SAS Code Debugger: User's Guide*
- *SAS Model Implementation Platform: User's Guide*

CONTACT INFORMATION

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

Jenna Austin
SAS Institute Inc.
Jenna.Austin@sas.com
http://www.sas.com

Toshiba Burns-Johnson
SAS Institute Inc.
Toshiba.Burns-Johnson@sas.com
http://www.sas.com

Aaron Mays
SAS Institute Inc.
Aaron.Mays@sas.com
http://www.sas.com

Mike Whitcher
SAS Institute Inc.
Mike.Whitcher@sas.com
http://www.sas.com