Chapter 5

Chapter Title: Macro Programming

ORCID:  0000-0002-7233-1461

Name:   A. John Bailer

Abstract:

This chapter introduces macros, programs that can be used to write programs, and macro variables. Macro code is processed before the SAS compiler is invoked. It is a powerful text processing facility that can be used to assist with writing SAS programs. Initially, macro variables are introduced as a simple way to generalize programs where much of the code will be reused with minor changes. Macros are code modules that can be used to generate SAS code based on parameters. Conditional execution and looping in macro programs are introduced along with options for debugging macro programs are introduced.

# 5.1 What Is a Macro and Why Would You Use It?

The macro processor is a text processor built into SAS. You should familiarize yourself with the macro processor because with it, you can execute the same programs with minor modification. A common workflow when developing a macro is to write a program, realize you want to use it again with minor modification and then you write a macro that does this. We start with a DATA step programming exercise that constructs a numerical estimate of the area under a standard normal density. In this exercise, macro coding and logic are used. Macro variables are substituted for hardcoded variables, and a macro is wrapped around the code for general use. This chapter provides information about debugging macros, storing macros, and using general built-in macro functions. The intent of this chapter is to provide you with an idea of why macros should be included in your toolbox. To learn more about macros, see Burlew (2014), Carpenter (2016), or the SAS Certification Prep Guide: Advanced Programming for SAS*®*9 (2014).

# 5.2 Motivation for Macros: Numerical Integration to Determine P(0<Z<1.645)

To motivate you to use macros, this section revisits a problem described in Chapter 3, “Case Study 2: Monte Carlo Integration to Estimate an Integral.” Or, in other words, how do you estimate P(0<Z<1.645)? In Chapter 3, a Monte Carlo simulation strategy is used. In this section, a numerical integration technique—the trapezoidal rule—is used. The trapezoidal rule approximates the area under a curve, f(x), over specified limits of integrations (say, low and high), by summing the areas of a collection of adjacent trapezoids constructed for a collection of points. For example:

[x1,f(x1)] [x2,f(x2)], … , [xk,f(xk)]

x*1*=low<x*2*< … < x*k-1* < x*k*=high

For more information about numerical integration techniques, see Burden and Faires (1989). If xi+1-xi=h for i=1,…,k-1, then the estimated area can be written as the following:



This area estimator is implemented in the program in Program 5.1, where the following is true:

k=25, x*1*=0, x*k*=1.645, h=(x*k*-x*1*)/24=1.645/24  


Program 5.1 Estimating P(0<Z<1.645) using the trapezoidal rule

**data** trapper;

trapsum=**0**;

array x\_value(**25**) x1-x25;

array f\_value(**25**) y1-y25;

low = **0**;

high = **1.645**;

incr = (high-low)/**24**;

multiplier = **1**/sqrt(**2**\*CONSTANT('PI'));

do point = **1** to **25**;

x\_value[point] = low + incr\*(point-**1**);

f\_value[point] = multiplier\*exp(-x\_value[point]\*x\_value[point]/**2**);

if point=**1** or point=**25** then trapsum = trapsum + f\_value[point]/**2**;

else trapsum = trapsum + f\_value[point];

end;

area\_est = trapsum\*incr;

output;

**run**;

**proc** **print** data=trapper noobs;

title "Trapezoidal Rule Area Estimate for P(0<Z<1.645)";

var low high incr area\_est;

**run**;

**data** trapper2;

set trapper;

array x\_value(**25**) x1-x25;

array f\_value(**25**) y1-y25;

do point =**1** to **25**;

xout = x\_value[point];

yout = f\_value[point];

output;

end;

**run**;

**proc** **print** data=trapper2 noobs;

title "Interpolation Points for Trapezoidal Rule";

var point low high incr area\_est xout yout;

**run**;

**proc** **sgplot** data=trapper2;

title "Plot of function values vs. x-values";

scatter x=xout y=yout;

**run**;

Program 5.1 generates and outputs the [xi,f(xi)] pairs based on x-values equally spaced over the limits of integration and the function evaluated at each x-value. It calculates the trapezoidal-rule area estimate, prints various elements of this analysis, and then plots the function values. The estimated area under the standard normal density is 0.44995 as shown in Table 5.1.

Table 5.1 Estimated P(0<Z<1.645)

| **low** | **high** | **incr** | **area\_est** |
| --- | --- | --- | --- |
| 0 | 1.645 | 0.068542 | 0.44995 |

The 25 points are shown in Table 5.2, which includes the common values of the limits of integration (**low** and **high**), the increment between consecutive points in the approximation (**incr**), the index of points (**point**), and the area estimate (**area\_est**).

Table 5.2 Estimated P(0<Z<1.645) and function values

| **point** | **low** | **high** | **incr** | **area\_est** | **xout** | **yout** |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1.645 | 0.068542 | 0.44995 | 0.00000 | 0.39894 |
| 2 | 0 | 1.645 | 0.068542 | 0.44995 | 0.06854 | 0.39801 |
| 3 | 0 | 1.645 | 0.068542 | 0.44995 | 0.13708 | 0.39521 |
| 4 | 0 | 1.645 | 0.068542 | 0.44995 | 0.20563 | 0.39060 |
| 5 | 0 | 1.645 | 0.068542 | 0.44995 | 0.27417 | 0.38423 |
| 6 | 0 | 1.645 | 0.068542 | 0.44995 | 0.34271 | 0.37619 |
| 7 | 0 | 1.645 | 0.068542 | 0.44995 | 0.41125 | 0.36659 |
| 8 | 0 | 1.645 | 0.068542 | 0.44995 | 0.47979 | 0.35557 |
| 9 | 0 | 1.645 | 0.068542 | 0.44995 | 0.54833 | 0.34326 |
| 10 | 0 | 1.645 | 0.068542 | 0.44995 | 0.61688 | 0.32982 |
| 11 | 0 | 1.645 | 0.068542 | 0.44995 | 0.68542 | 0.31542 |
| 12 | 0 | 1.645 | 0.068542 | 0.44995 | 0.75396 | 0.30024 |
| 13 | 0 | 1.645 | 0.068542 | 0.44995 | 0.82250 | 0.28445 |
| 14 | 0 | 1.645 | 0.068542 | 0.44995 | 0.89104 | 0.26823 |
| 15 | 0 | 1.645 | 0.068542 | 0.44995 | 0.95958 | 0.25174 |
| 16 | 0 | 1.645 | 0.068542 | 0.44995 | 1.02813 | 0.23517 |
| 17 | 0 | 1.645 | 0.068542 | 0.44995 | 1.09667 | 0.21865 |
| 18 | 0 | 1.645 | 0.068542 | 0.44995 | 1.16521 | 0.20234 |
| 19 | 0 | 1.645 | 0.068542 | 0.44995 | 1.23375 | 0.18637 |
| 20 | 0 | 1.645 | 0.068542 | 0.44995 | 1.30229 | 0.17086 |
| 21 | 0 | 1.645 | 0.068542 | 0.44995 | 1.37083 | 0.15590 |
| 22 | 0 | 1.645 | 0.068542 | 0.44995 | 1.43938 | 0.14159 |
| 23 | 0 | 1.645 | 0.068542 | 0.44995 | 1.50792 | 0.12798 |
| 24 | 0 | 1.645 | 0.068542 | 0.44995 | 1.57646 | 0.11515 |
| 25 | 0 | 1.645 | 0.068542 | 0.44995 | 1.64500 | 0.10311 |

The graphic produced by Program 5.1 is shown in Figure 5.1. Questions: Can you modify this program so that vertical lines connect the density values to the x-axis? Can you connect the density values to display the trapezoids being summed to obtain the estimated area? As a final enhancement, can you superimpose the true density?

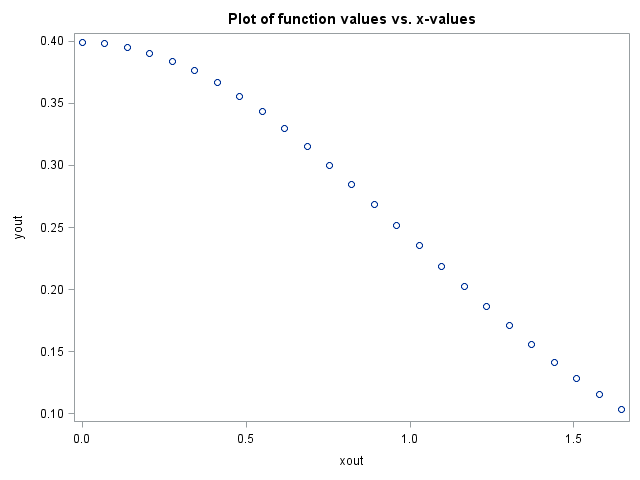


Figure 5.1 Plot of the [x,f(x)] values used in the trapezoidal rule

You have a specific implementation for a particular problem. What can you do with this implementation? Consider the following questions:

1. How hard is it to modify this code if you want *k*=50 points versus *k*=25 points?
2. Can you generalize this code to specify different limits of integration?
3. Can you set up a switch to produce graphics only when they are requested?
4. Is it possible to package this estimator as a callable program, subroutine, or function?
5. Can you generalize this code for any arbitrary function?

For the impatient, the answers to these questions are: not too hard, easily, easily, yes, and yes. The common denominator in all of these answers is macros.

# 5.3 Processing Macros

“You talkin’ to me? You talkin’ to me? You talkin’ to me?” says Travis Bickle in the film Taxi Driver.

The macro processor is the Travis Bickle of SAS. To understand why, look at how SAS processes a program in the following short overview. If you want a more detailed description, see Introduction to SAS Programs and Macro Processing at support.sas.com, or Burlew (2014).

When program statements are submitted, they are sent to the input stack, which is an area of computer memory. The word scanner processes these program statements by converting them to tokens. Tokens are directed to an appropriate location for additional processing. Appropriate locations include the DATA step compiler, command processor, and macro processor. If a program contains any macro statements, then SAS sets up a symbol table for the macro variables. Actually, SAS defines certain macro variables every time you use it. For example, the day that you run SAS is a value stored in a macro variable named SYSDAY. SAS needs to resolve any macro requests that are part of a series of commands before these commands are processed.

How does the word scanner know how to direct tokens to an appropriate location? Any non-blank text that is preceded by a percentage sign (%) or ampersand (&) is a macro trigger. When the word scanner encounters a macro trigger, it sends the tokens with % or & to the macro processor. (So, “you are talkin’” to the macro processor if you are text that contains a macro trigger.) The macro processor interacts with the symbol table to store or extract the values of macro variables. Or, more generally, the macro processor interacts with the word scanner to resolve any macro statements. The macro processor substitutes the text demanded of it from the macro statements into the input stack. This continues until the input stack has been completely processed. All of this will become clearer when you look at how code with macro statements and variables is resolved.

Worth Noting: Percentage sign (%) and ampersand (&) are special triggers to direct activity to the macro processor in SAS before compiling and execution in SAS.

# 5.4 Macro Variables, Parameters, and Functions

A macro variable can be referenced anywhere in a SAS program. The only exception is in data lines that are being read into SAS. A macro variable can be assigned text values. The value of a variable is stored in either a local or global symbol table. A macro variable name must be a valid SAS name. (#$#%@#$%#!! won’t work, even though you might feel this way about a programming project.) A macro variable is preceded by an & when it is referenced in a SAS program. When a macro variable is resolved in a program, it might be located near other text. If it follows text, then it is referenced as text&macro-variable. If it precedes text, then it is referenced with a period delimiting the value from the text (&macro-variable.text). SAS can define certain macro variables (automatic macro variables), while a program can define other macro variables (user-defined macro variables.)

Worth Noting: When a macro variable is stored globally, its value can be accessed anywhere in a SAS program. Which a macro variable is stored locally, its value is only available in a macro.

When you first start macro programming in SAS, you will most likely use macro variables to define simple substitutions in a program. In Program 5.2, the trapezoidal numerical integration routine is modified to address the previous questions.

1. How hard is it to modify this code if you want *k*=50 points versus *k*=25 points?

Can you generalize this code to specify different limits of integration?

The macro statement %let assigns values to npts(the macro variable for k), and to low and high(the macro variables that correspond to the limits of integration). In the following code, every occurrence of 25 in the program is replaced with &npts. Every occurrence of 24 is replaced with &npts-1. Because you can specify any arbitrary number of points to be evaluated in this numerical integration, the first question is answered. Including macro variables &LOW and &HIGH generalizes the code to specify different limits of integration. As a result, the second question is answered. These macro variables are incorporated into titles in this modified program. Additional bells and whistles are added, including using a put statement to write the results, and using a file statement to specify a location for the putoutput.

Worth Noting: %LET is used to directly define a macro variable for use in a program. You will often start macro-izing a program by replacing hard coded values with macro variables and assigning values to these macro variables with %LET statements.

Program 5.2 Replacing the limits of integration and number of points with macro variables

%let Folder=C:\Users\baileraj\Documents\book-SPiS-2nd-ed;

%let subFolder=chapter05;

%let npts = 50; \* 25, 15, 5, and 3 tried as well;

%let LOW = -1.645;

%let HIGH = 1.645;

**data** trapper;

file "&Folder\&subFolder\ch5-fig5.5-est.out" MOD;

trapsum = **0**;

array x\_value(&npts) x1-x&npts;

array f\_value(&npts) y1-y&npts;

low = &LOW;

high = &HIGH;

incr = (high-low)/( &npts -**1**);

multiplier = **1**/sqrt(**2**\*CONSTANT('PI'));

do point = **1** to &npts;

x\_value[point] = low + incr\*(point-**1**);

f\_value[point] = multiplier\*exp(-x\_value[point]\*x\_value[point]/**2**);

if point =**1** or point =&npts then trapsum = trapsum + f\_value[point]/**2**;

else trapsum = trapsum + f\_value[point];

end;

area\_est = trapsum\*incr;

output;

put;

put "est. P(&LOW < Z < &HIGH) =" area\_est "(based on &NPTS points)";

put;

**run**;

**proc** **print** data=trapper noobs;

title "Trapezoidal Rule Area Estimate for P(&LOW<Z<&HIGH)";

title2 "(based on &NPTS equally spaced points)";

var low high incr area\_est;

**run**;

**data** trapper2;

set trapper;

array x\_value(&npts) x1-x&npts;

array f\_value(&npts) y1-y&npts;

do point=**1** to &npts;

xout = x\_value[point];

yout = f\_value[point];

output;

end;

**run**;

**proc** **print** data=trapper2 noobs;

title "Interpolation Points for Trapezoidal Rule";

var point low high incr area\_est xout yout;

**run**;

**proc** **sgplot** data=trapper2;

title "Plot of function values vs. x-values";

scatter x=xout y=yout;

**run**;

When you run this program, you generate the estimate of the area between -1.645 and 1.645 under a standard normal curve, see Table 5.3. "

Worth Noting: %LET with macro variables have been used to define file folders and subfolders in the production of output for this text. The statement

file "&Folder\&subfolder\est.out" MOD;

identifies a file location for the results of a put statement and the MOD option specifies the addition of new lines of output if the file already exists.

Table 5.3 Trapezoidal rule estimate of P(-1.645<Z<1.645)

**Trapezoidal Rule Area Estimate for P(-1.645<Z<1.645)**

**(based on 50 equally spaced points)**

| **low** | **high** | **incr** | **area\_est** |
| --- | --- | --- | --- |
| -1.645 | 1.645 | 0.067143 | 0.89990 |

If you repeatedly run this program and change &npts to different values, then you get the output in Table 5.4.

Figure 5.4 Output written to the file ch5-fig5.5-est.out

est. P(-1.645 < Z < 1.645) =0.8999027424 (based on 50 points)

est. P(-1.645 < Z < 1.645) =0.8994989035 (based on 25 points)

est. P(-1.645 < Z < 1.645) =0.8984685898 (based on 15 points)

est. P(-1.645 < Z < 1.645) =0.8808617859 (based on 5 points)

est. P(-1.645 < Z < 1.645) =0.8258773355 (based on 3 points)

Table 5.4 is generated by five distinct SAS code submissions. Each submission involves reassigning a value of &npts. It would be great to loop over a range of &npts values as part of this code. SAS macro programming enables you to do this using the macro statements %DO and %END (equivalents of the DATA step statements DO-END). You can automate this process to a greater degree by taking this code where macro variables have replaced hard-coded variables and embedding it in a macro, a collection of statements that are executed, possibly with specified input parameters that are macro variables.

# 5.5 Conditional Execution, Looping, and Macros

The %LET macro statement can be used anywhere in a program in open code. Other macro statements, such as %DO and %END and %IF and %THEN, can be used only in the context of a defined macro. A macro is a collection of commands that are evaluated by the macro processor. The general construction of a macro involves declaring the macro and any variables that might be passed to it, specifying the statements that are to be invoked in the macro, and closing with a statement that indicates the end of the macro. A macro program looks like the following:

%macro prog-name;

<text line1 >

<text line2 >

<text line3 >

<. . . >

%mend prog-name;

A macro program can be viewed as a subroutine or function, and it is invoked by a reference to its name with a special macro trigger, e.g. %PROG-NAME. The semicolon is not needed. %PROG-NAME is “talking” to the macro processor, not to other SAS components. Parameters can be passed to macro programs.

WORTH NOTING: The word “parameter” has a technical meaning to the statistical community, and a different meaning to the programming community. Here, a “parameter” is an argument to a macro whose value might influence what a macro does.

In the trap\_area\_Z macro defined in Program 5.3, a macro wrapper is placed around the trapezoidal rule area estimator code. Parameters are defined for this macro: LOW, HIGH, npts\_lo, npts\_hi, npts\_by, fout, print\_est, print\_pts, display\_graph, and ODS\_on. The macro declaration for trap\_area\_Z includes these parameters and their default values.

**%macro** trap\_area\_Z(LOW=-**1.645**, HIGH=**1.645**, npts\_lo=**10**, npts\_hi=**10**,

npts\_by=**2**,fout=&Folder\&subFolder\est3.out,

print\_est=FALSE, print\_pts=FALSE, display\_graph=FALSE, ODS\_on=FALSE);

Worth Noting: A macro is defined by %macro statement followed by the name of the macro. Parameters can be given a default value in the declaration of a macro. For example, %macro trap\_area\_Z(LOW=-1.645,…) has 10 parameters defined with defaults set for each including -1.645 being the default for the macro parameter LOW.

Keyword parameters are used in the trap\_area\_Z macro definition. Parameters can be passed by position, by keyword, or by using a mix of the two strategies. (If a mix is used, then positional parameters must be specified before keyword parameters.) If you define a keyword parameter, then you must call the macro with a keyword in the argument list. The style used in the macro definition needs to match the style used when invoking the macro. In the spirit of writing self-documenting code, keyword parameters are preferred.

The construction of the trap\_area\_Z macro explicitly addresses two questions from Section 5.2:

1. Can you set up a switch to produce graphics only when they are requested?
2. Is it possible to package this estimator as a callable program, subroutine, or function?

Let’s start with the name of this macro and its input parameters. The input parameters are assigned default values in the macro. Macro comments describe the input parameters. These comments begin with a %\* and end with a semicolon. If you use standard comments (e.g., you start with an asterisk and end with a semicolon), the comments are displayed in the log. However, comments are not displayed until the appropriate options are set. Note: The preferred and safest comments to use inside a macro are of the form /\* comment … \*/.

In Program 5.2, you converted hardcoded values into macro variables. The macro variables were assigned values using %LET statements. In the following macro, printing, generating graphs, and using ODS RTF are all based on parameters of the macro.

Worth Noting: The parameters of a macro are referenced as local macro variables in the macro. Thus, the parameter LOW in %macro trap\_area\_Z(LOW=-1.645 …) is referenced in the macro as &LOW. The macro variable is defined only in the macro, hence it is called a local macro variable.

The %IF and %THEN statements check whether graphics are requested. For example, submit the following block of code:

%if %upcase(&display\_graph)=TRUE %then %do;

proc sgplot data=trapper2;

title "Plot of function values vs. x-values";

scatter x=xout y=yout;

run;

%end;

This code causes the following lines to be included in the input stack for processing when the macro parameter display\_graph=TRUE,

proc sgplot data=trapper2;

title "Plot of function values vs. x-values";

scatter x=xout y=yout;

run;

The %upcase function guarantees that a &display\_graph value of TRUE, True, true, TrUe (you get the idea) is converted to TRUE before the value is compared with the condition. The %IF and %THEN statements are then evaluated by the macro processor. If true, then the code between the %DO and %END statements is processed as part of the input stack. The macro can evaluate the integral based on a range of values. This option is set using %DO, %TO, and %BY with a %END statement. For example:

%do npts=&npts\_lo %to &npts\_hi %by &npts\_by; \*loop over npts values;

Worth Noting: It is easy to get confused when programming conditional statements (IF-THEN) or looping (DO-END) in DATA step programming vs. conditional statements (%IF-%THEN) or looping (%DO-%END) in macro programming. The critical question is are you programming for the macro processor, a task that writes SAS code for later execution by the SAS compiler, or for the SAS compiler.

If you put all of these ideas together, you get the macro shown in Program 5.3.

Program 5.3 Macro implementing a trapezoidal rule estimate of the area under a standard normal density curve

**%macro** trap\_area\_Z(LOW=-**1.645**, HIGH=**1.645**, npts\_lo=**10**, npts\_hi=**10**, npts\_by=**2**,

fout=&Folder\&subFolder\est3.out,

print\_est=FALSE, print\_pts=FALSE, display\_graph=FALSE, ODS\_on=FALSE);

/\* ========================================================================

Purpose: Estimate P{LOW<Z< HIGH) using the trapezoidal rule

Macro variables:

LOW, HIGH: interval of interest

NPTS\_LO, NPTS\_HI, NPTS\_BY: # function values evaluated in area calc.

FOUT: output data file containing area estimate for each NPTS value

PRINT\_EST: print area estimate

PRINT\_PTS: print points/nodes {x1-xn} + function values {f(x1)-f(xn)}

DISPLAY\_GRAPH: generate PROC GPLOT with function values

ODS\_ON: generate ODS RTF output

========================================================================

\*/

%do npts = &npts\_lo %to &npts\_hi %by &npts\_by; \*loop over npts values;

data trapper;

file "&fout" MOD;

trapsum = **0**;

array x\_value(&npts) x1-x&npts;

array f\_value(&npts) y1-y&npts;

low = &LOW;

high = &HIGH;

incr = (high-low)/( &npts -**1**);

multiplier = **1**/sqrt(**2**\*CONSTANT('PI'));

do point = **1** to &npts;

x\_value[point] = low + incr\*(point-**1**);

f\_value[point] = multiplier\*exp(-x\_value[point]\*x\_value[point]/**2**);

if point=**1** or point=&npts then trapsum = trapsum + f\_value[point]/**2**;

else trapsum = trapsum + f\_value[point];

end;

area\_est = trapsum\*incr;

output;

put "est. P(&LOW < Z < &HIGH) =" area\_est "(based on &NPTS points)";

run;

%if %upcase(&ODS\_ON)=TRUE %then %do;

ods rtf file="&Folder\&subFolder\ch5-fig5.6.rtf"

image\_dpi=**300**

style=sasuser.customSapphire;

%end;

%if %upcase(&print\_est)=TRUE %then %do;

proc print data=trapper;

title "Trapezoidal Rule Area Estimate for P(&LOW<Z<&HIGH)";

title2 "(based on &NPTS equally spaced points)";

var low high incr area\_est;

run;

%end;

data trapper2;

set trapper;

array x\_value(&npts) x1-x&npts;

array f\_value(&npts) y1-y&npts;

do point = **1** to &npts;

xout = x\_value[point];

yout = f\_value[point];

output;

end;

run;

%if %upcase(&print\_pts)=TRUE %then %do;

proc print data=trapper2;

title "Interpolation Points for Trapezoidal Rule";

var point low high incr area\_est xout yout;

run;

%end;

%if %upcase(&display\_graph)=TRUE %then %do;

proc sgplot data=trapper2;

title "Plot of function values vs. x-values";

scatter x=xout y=yout;

run;

%end;

%if %upcase(&ODS\_ON)=TRUE %then %do;

ods rtf close;

%end;

%end; \* of loop over npts values;

**%mend** trap\_area\_Z;

Once the macro trap\_area\_Z is defined, this can be invoked by typing %trap\_area\_Z with particular parameters defined. Consider the following invocations of the macro:

* %trap\_area\_Z(): invokes the macro with all of the default settings, namely, LOW=-1.645, HIGH=1.645, npts\_lo=10, npts\_hi=10, npts\_by=2, fout=&Folder\&subFolder\est3.out, print\_est=FALSE, print\_pts=FALSE, display\_graph=FALSE, ODS\_on=FALSE.
* %trap\_area\_Z(LOW=0): invokes the macros with all defaults except LOW=0
* %trap\_area\_Z(LOW=-.67, HIGH=.67): change both limits of integration
* %trap\_area\_Z(LOW=-.67, HIGH=.67, npts\_lo=10, npts\_hi=20, npts\_by=5): changes limits of integration and uses 10, 15 and 20 points in constructing trapezoids for the estimate
* %trap\_area\_Z(LOW=-.67, HIGH=.67, print\_est=TRUE, print\_pts=true): estimates the area between -0.67 and 0.67 and prints the estimated area (print\_est=TRUE parameter) and the interpolation points (print\_pts=true parameter)
* %trap\_area\_Z(LOW=-.67, HIGH=.67, display\_graph=TRUE): estimates the area over the interval [-0.67, 0.67] and plots the interpolation points
* %trap\_area\_Z(LOW=-.67, HIGH=.67, ODS\_ON=TRUE, display\_graph=TRUE): estimates the area over the interval [-0.67, 0.67] and saves the plot of the interpolation points in an RTF file

How would you modify the macro to specify the name of the RTF file as a macro parameter? How would you modify the macro to do validity and range checks (e.g. HIGH > LOW)?

## More complicated macro variable construction

There will be times when combining two macro variables to reference another macro variable might be useful. In the example in Program 5.3, two macro variables need to be resolved to define a third macro variable. Suppose observations have variables containing hypothetical values of weights (weight1 and weight2) measured on two separate occasions (week1 and week2). In this constructed data set, a weight measurement at week 15 was 70 (week1=15, weight1=70) and a weight measurement at week 25 was 74 (week2=25, weight2=74).

Two macro parameters (variable and obs) identify the variable and occasion of interest. This macro is designed to echo the input to the log and print the value of the variable of interest.

Program 5.4 Resolving a complex macro variable

%let var1=week;

%let var2=weight;

%let time1=1;

%let time2=2;

%let var1time1 = week1;

%let var1time2 = week2;

%let var2time1 = weight1;

%let var2time2 = weight2;

data tester;

input week1 weight1 week2 weight2;

datalines;

15 70 25 74

;

run;

**%macro** showvalue(variable, obs);

%put Value of '&variable' = &variable;

%put Value of '&obs' = &obs;

%put Value of '&&&variable.&obs' = AMPERSAND-&variable.&obs = &&&variable.&obs;

data \_NULL\_;

set tester;

put "Value of &&&variable.&obs =" &&&variable.&obs;

run;

**%mend** showvalue;

%***showvalue***(variable=var1, obs=time1)

%***showvalue***(variable=var2, obs=time2)

An interesting construct in this program is &&&variable.&obs. Macro variable names are resolved from left to right. && is resolved to & by the macro processor. In this example, when &variable=var1(=week) and &obs=time1(=1), &&&variable.&obs has the value var1time1(=week1). The SAS log from invoking this macro is shown in Table 5.5.

Worth Noting: Macro variable names are evaluated from left to right and && resolves to & by the macro processor. The macro processor will take multiple passes through a more complex macro variable name until the macro variable is fully processed.

Table 5.5 SAS LOG Output from invoking the macro

Value of '&variable' = var1

Value of '&obs' = time1

Value of '&&&variable.&obs' = AMPERSAND-var1time1 = week1

Value of week1 =15

Value of '&variable' = var2

Value of '&obs' = time2

Value of '&&&variable.&obs' = AMPERSAND-var2time2 = weight2

Value of weight2 =74

## Changing locations in a macro during execution

You can direct the macro processor to move to a different location during its execution using %GOTO. %GOTO is especially handy when checking for errors. In the code in Program 5.5 and in the output in Table 5.6, an error message is generated if an invalid argument is passed as a parameter. The %sysevalf function evaluates whether &npts<1. If it is, an error message is generated and a branch is executed. The macro variable &npts is enclosed in single quotation marks. This keeps the macro processor from evaluating this variable. If the macro variable is enclosed in double quotation marks, then the macro processor substitutes the value of the macro variable in the expression.

Worth Noting: A macro variable enclosed in single quotes, e.g. '&variable', is not evaluated by the macro processor while a macro variable enclosed in double quotes, e.g. "&variable", is evaluated. Referencing a macro variable with double quotes is particularly useful when building a title that includes values of the macro variables as used in the trap\_area\_Z macro.

**Program 5.5 Macro illustrating %GOTO**

%macro ncheck(npts);

%\* npts = needs to be greater than or equal to 1;

%if %sysevalf(&npts<1) %then %do;

%put ERROR: '&npts' must exceed 1;

%put ERROR: value of '&npts' = &npts;

%goto badend;

%end;

%put Value of '&npts' = &npts;

%badend: ;

%mend ncheck;

%ncheck(1)

%ncheck(-2)

%ncheck(0.5)

The output is shown in Table 5.6.

Table 5.6 NCHECK macro applied with three different arguments

%ncheck(1)

Value of '&npts' = 1

%ncheck(-2)

ERROR: '&npts' must exceed 1

ERROR: value of '&npts' = -2

%ncheck(0.5)

ERROR: '&npts' must exceed 1

ERROR: value of '&npts' = 0.5

An alternative is to check if &NPTS<1, and then use an %ELSE branch. This is preferred over %GOTO in programming.

This section started with five questions. The fifth question has not been addressed.

Can you generalize this code for any arbitrary function?

This section does not provide the answer. However, you should try it at home. Where in the code is the standard normal density function defined? Can you define a macro variable that contains this arbitrary function, and then pass it as a parameter to a general TRAP\_AREA\_Z macro? Do you need to change more than one line in the previous macro? If you were producing this as a macro that was going to be used in a production environment, would you need to add checks for valid parameter values? How would you do this in the TRAP\_AREA\_Z macro?

Now, the claim could be made that this code matches the first attempt at constructing this macro. Alas, the truth is that debugging and error correction are required. Basic strategies for exploring and debugging errors in macro coding are addressed in the next section.

# 5.6 Debugging Macro Code and Programs

## Write out Values of Macro Variables

The first thing to do with non-working macro code is to write out the values of the macro variables. You can write to the log the value of a macro variable that you have defined using the statement %PUT &MACRO-VAR-NAME.

WORTH NOTING: You can display SAS automatic macro variables (%PUT \_AUTOMATIC\_), user-defined macro variables (%PUT \_USER\_), or both types (%PUT \_ALL\_).

To debug, specifying a particular macro variable or all user-defined macro variables is more useful. It might be interesting to determine what macro variables are defined by SAS. You might even be interested in using some of these variables in your programs. The SYSDATE, SYSDAY, and SYSTIME macro variables are useful for extracting date and time information. SYSLAST is useful for defining the use of a default data set in a macro. Program 5.6 requests the user-defined, automatically-defined and all macro variable value.

Program 5.6 Requesting the display of the values of SAS user, automatic and all macro variables

%put \_USER\_;

%put \_AUTOMATIC\_;

%put \_ALL\_;

These values are all written to the SAS LOG. An edited portion of this LOG output is presented in Table 5.7. The \_USER\_ macro variables includes the names of the folder and subfolder for this chapter along with values of macros variables defined in programs run before this program. Note that these are all GLOBAL.

Table 5.7 Displaying Values of Macro Variables

353 %put \_USER\_;

GLOBAL FOLDER C:\Users\baileraj\Documents\book-SPiS-2nd-ed

GLOBAL HIGH 1.645

GLOBAL LOW -1.645

GLOBAL NPTS 12

GLOBAL SUBFOLDER chapter05

GLOBAL TIME1 1

GLOBAL TIME2 2

GLOBAL VAR1 week

GLOBAL VAR1TIME1 week1

GLOBAL VAR1TIME2 week2

GLOBAL VAR2 weight

GLOBAL VAR2TIME1 weight1

GLOBAL VAR2TIME2 weight2

354 %put \_AUTOMATIC\_;

AUTOMATIC AFDSID 0

AUTOMATIC AFDSNAME

…

AUTOMATIC SYSDATE 18JUN18

AUTOMATIC SYSDATE9 18JUN2018

AUTOMATIC SYSDAY Monday

…

AUTOMATIC SYSVER 9.4

AUTOMATIC SYSVLONG 9.04.01M4P110916

AUTOMATIC SYSVLONG4 9.04.01M4P11092016

AUTOMATIC SYSWARNINGTEXT

The macro variables that are automatically defined include SAS system information such as the version of SAS and information connected to the session such as the date when the program was run. The information produced by %put \_ALL\_ is the union of the output produced by %put \_USER\_ and %put \_AUTOMATIC\_.

## Useful SAS Options for Debugging Macros

There are three main options that are important when constructing and debugging macros. The SYMBOLGEN option shows the values of macro variables. It can be used to trace the resolution of complex macro variable names. The MPRINT option displays the SAS statements that are generated for execution by the macro processor. The MLOGIC option displays a trace of the macro processor’s execution of the macro program. It might be the most useful of all three. The next three displays show the output of these options. Program 5.7 invokes each of these options in turn with trap\_area\_z.

WORTH NOTING Options can be set in SAS that apply for the duration of the program (or until other options are set). Options can control output or request information be written to the SAS LOG as we see in this section.

Program 5.7 Illustrating output produced by SAS options for debugging macros

options mprint; \* turn on MPRINT;

%***trap\_area\_z***(LOW=**0**,HIGH=**1.96**,npts\_lo=**15**,npts\_hi=**25**,npts\_by=**5**, display\_graph=TRUE)

options nomprint symbolgen; \* turn off MPRINT, turn on SYMBOLGEN;

%***trap\_area\_z***(LOW=**0**,HIGH=**1.96**,npts\_lo=**15**,npts\_hi=**25**,npts\_by=**5**, display\_graph=TRUE)

options nosymbolgen mlogic; \* turn off SYMBOLGEN, turn on MLOGIC;

%***trap\_area\_z***(LOW=**0**,HIGH=**1.96**,npts\_lo=**15**,npts\_hi=**25**,npts\_by=**5**, display\_graph=TRUE)

options nomlogic; \* turn off MLOGIC;

Consider the information written to the SAS LOG after invoking the trap\_area\_z macro with each of these options turned on. Table 5.9 displays what is produced by the MPRINT option for the npts = 15 case. Note that the messages related to data sets created and execution time were omitted from this Table

Table 5.8 SAS LOG (partial) from Using Option MPRINT with trap\_area\_Z invocation

543 options mprint; \* turn on MPRINT;

544 %trap\_area\_z(LOW=0,HIGH=1.96,npts\_lo=15,npts\_hi=25,npts\_by=5, display\_graph=TRUE)

MPRINT(TRAP\_AREA\_Z): \*loop over npts values;

MPRINT(TRAP\_AREA\_Z): data trapper;

MPRINT(TRAP\_AREA\_Z): file "C:\Users\baileraj\Documents\book-SPiS-2nd-ed\chapter05\est3.out" MOD;

MPRINT(TRAP\_AREA\_Z): trapsum = 0;

MPRINT(TRAP\_AREA\_Z): array x\_value(15) x1-x15;

MPRINT(TRAP\_AREA\_Z): array f\_value(15) y1-y15;

MPRINT(TRAP\_AREA\_Z): low = 0;

MPRINT(TRAP\_AREA\_Z): high = 1.96;

MPRINT(TRAP\_AREA\_Z): incr = (high-low)/( 15 -1);

MPRINT(TRAP\_AREA\_Z): multiplier = 1/sqrt(2\*CONSTANT('PI'));

MPRINT(TRAP\_AREA\_Z): do point = 1 to 15;

MPRINT(TRAP\_AREA\_Z): x\_value[point] = low + incr\*(point-1);

MPRINT(TRAP\_AREA\_Z): f\_value[point] = multiplier\*exp(-x\_value[point]\*x\_value[point]/2);

MPRINT(TRAP\_AREA\_Z): if point=1 or point=15 then trapsum = trapsum + f\_value[point]/2;

MPRINT(TRAP\_AREA\_Z): else trapsum = trapsum + f\_value[point];

MPRINT(TRAP\_AREA\_Z): end;

MPRINT(TRAP\_AREA\_Z): area\_est = trapsum\*incr;

MPRINT(TRAP\_AREA\_Z): output;

MPRINT(TRAP\_AREA\_Z): put "est. P(0 < Z < 1.96) =" area\_est "(based on 15 points)";

MPRINT(TRAP\_AREA\_Z): run;

MPRINT(TRAP\_AREA\_Z): data trapper2;

MPRINT(TRAP\_AREA\_Z): set trapper;

MPRINT(TRAP\_AREA\_Z): array x\_value(15) x1-x15;

MPRINT(TRAP\_AREA\_Z): array f\_value(15) y1-y15;

MPRINT(TRAP\_AREA\_Z): do point = 1 to 15;

MPRINT(TRAP\_AREA\_Z): xout = x\_value[point];

MPRINT(TRAP\_AREA\_Z): yout = f\_value[point];

MPRINT(TRAP\_AREA\_Z): output;

MPRINT(TRAP\_AREA\_Z): end;

MPRINT(TRAP\_AREA\_Z): run;

MPRINT(TRAP\_AREA\_Z): proc sgplot data=trapper2;

MPRINT(TRAP\_AREA\_Z): title "Plot of function values vs. x-values";

MPRINT(TRAP\_AREA\_Z): scatter x=xout y=yout;

MPRINT(TRAP\_AREA\_Z): run;

The SAS option MPRINT displays the SAS code that is produced by the macro processor. This is essentially the program that was written by the macro and will now be executed by the SAS compiler. It doesn’t have the attractive indenting and readability of the original code; however, it is great to see what the macro processor has produced. The results of turning on this SYMBOLGEN option is presented in Table 5.9.

Table 5.9 SAS LOG from Using Option SYMBOLGEN with trap\_area\_Z invocation

546 options nomprint symbolgen; \* turn off MPRINT, turn on SYMBOLGEN;

547 %trap\_area\_z(LOW=0,HIGH=1.96,npts\_lo=15,npts\_hi=25,npts\_by=5, display\_graph=TRUE)

SYMBOLGEN: Macro variable NPTS\_LO resolves to 15

SYMBOLGEN: Macro variable NPTS\_HI resolves to 25

SYMBOLGEN: Macro variable NPTS\_BY resolves to 5

SYMBOLGEN: Macro variable FOUT resolves to &Folder\&subFolder\est3.out

SYMBOLGEN: Macro variable FOLDER resolves to C:\Users\baileraj\Documents\book-SPiS-2nd-ed

SYMBOLGEN: Macro variable SUBFOLDER resolves to chapter05

SYMBOLGEN: Macro variable NPTS resolves to 15

SYMBOLGEN: Macro variable NPTS resolves to 15

SYMBOLGEN: Macro variable NPTS resolves to 15

SYMBOLGEN: Macro variable NPTS resolves to 15

SYMBOLGEN: Macro variable LOW resolves to 0

SYMBOLGEN: Macro variable HIGH resolves to 1.96

SYMBOLGEN: Macro variable NPTS resolves to 15

SYMBOLGEN: Macro variable NPTS resolves to 15

SYMBOLGEN: Macro variable NPTS resolves to 15

SYMBOLGEN: Macro variable LOW resolves to 0

SYMBOLGEN: Macro variable HIGH resolves to 1.96

SYMBOLGEN: Macro variable NPTS resolves to 15

The SAS option SYMBOLGEN displays the values that were assigned to the macro variables during the processing of the macro. Finally, the result of requesting the MLOGIC option is displayed in Table 5.10.

Table 5.10 SAS LOG from Using Option MLOGIC with trap\_area\_Z invocation

548 options nosymbolgen mlogic; \* turn off SYMBOLGEN, turn on MLOGIC;

549 %trap\_area\_z(LOW=0,HIGH=1.96,npts\_lo=15,npts\_hi=25,npts\_by=5, display\_graph=TRUE)

MLOGIC(TRAP\_AREA\_Z): Beginning execution.

MLOGIC(TRAP\_AREA\_Z): Parameter LOW has value 0

MLOGIC(TRAP\_AREA\_Z): Parameter HIGH has value 1.96

MLOGIC(TRAP\_AREA\_Z): Parameter NPTS\_LO has value 15

MLOGIC(TRAP\_AREA\_Z): Parameter NPTS\_HI has value 25

MLOGIC(TRAP\_AREA\_Z): Parameter NPTS\_BY has value 5

MLOGIC(TRAP\_AREA\_Z): Parameter DISPLAY\_GRAPH has value TRUE

MLOGIC(TRAP\_AREA\_Z): Parameter FOUT has value &Folder\&subFolder\est3.out

MLOGIC(TRAP\_AREA\_Z): Parameter PRINT\_EST has value FALSE

MLOGIC(TRAP\_AREA\_Z): Parameter PRINT\_PTS has value FALSE

MLOGIC(TRAP\_AREA\_Z): Parameter ODS\_ON has value FALSE

MLOGIC(TRAP\_AREA\_Z): %DO loop beginning; index variable NPTS; start value is 15; stop value is 25; by value is 5.

MLOGIC(TRAP\_AREA\_Z): %IF condition %upcase(&ODS\_ON)=TRUE is FALSE

MLOGIC(TRAP\_AREA\_Z): %IF condition %upcase(&print\_est)=TRUE is FALSE

MLOGIC(TRAP\_AREA\_Z): %IF condition %upcase(&print\_pts)=TRUE is FALSE

MLOGIC(TRAP\_AREA\_Z): %IF condition %upcase(&display\_graph)=TRUE is TRUE

WARNING: HTML4 destination does not support EPSI images. Using the default static format.

NOTE: There were 15 observations read from the data set WORK.TRAPPER2.

MLOGIC(TRAP\_AREA\_Z): %IF condition %upcase(&ODS\_ON)=TRUE is FALSE

MLOGIC(TRAP\_AREA\_Z): %DO loop index variable NPTS is now 20; loop will iterate again.

MLOGIC(TRAP\_AREA\_Z): %IF condition %upcase(&ODS\_ON)=TRUE is FALSE

MLOGIC(TRAP\_AREA\_Z): %IF condition %upcase(&print\_est)=TRUE is FALSE

MLOGIC(TRAP\_AREA\_Z): %IF condition %upcase(&print\_pts)=TRUE is FALSE

MLOGIC(TRAP\_AREA\_Z): %IF condition %upcase(&display\_graph)=TRUE is TRUE

The SAS option MLOGIC displays information about the looping and conditional execution steps in a macro along with the values that were assigned to the macro variables during the processing of the macro. For example, the starting, stopping and incremental values of the loop index NPTS is echoed:

MLOGIC(TRAP\_AREA\_Z): %DO loop beginning; index variable NPTS;

start value is 15; stop value is 25; by value is 5.

In addition, the check of whether ODS is requested, i.e. the condition in the %IF statement, %IF %upcase(&ODS\_ON)=TRUE is evaluated – MLOGIC reports this condition is FALSE and no additional code is produced by the macro processor. The request to print the results, %upcase(&print\_est)=TRUE, also is evaluated to be FALSE. The %IF check to display the plot, %IF condition %upcase(&display\_graph)=TRUE is TRUE and the requested plot is produced.

Try this at home: redo this code with both MPRINT and MLOGIC requested and examine the results.

# 5.7 Saving Macros

The previous examples used macros shortly after they were defined. You are often going to want to save macros for future use. Introduction to Storing and Reusing Macros at support.sas.com provides a good foundation for learning more about saving macros. Three general strategies for using a previously defined macro include the following:

1. Insert the macro code from an external file.
2. Use the autocall facility in SAS.

Use a stored and compiled macro.

For the first strategy, the %INCLUDE command inserts the contents of an external file into the location from which the %INCLUDE command is issued. For the second strategy, the autocall facility collects a set of macros as external text (.sas) files in a specified folder. A macro in the autocall facility is stored with a filename that matches the actual macro name. The macros called using the autocall facility are compiled and stored in WORK.SASMCR when they are first called in a SAS session. The option MAUTOSOURCE must be set to use the autocall facility, and the SASAUTOS option must point to where the autocall facility macro is stored. For the third strategy, compiled macros are stored in a SAS catalog. The options MSTORED and SASMSTORE are used. In addition, the STORE option must be set in the macro definition. This strategy is preferred if you have macros that are production quality and are not frequently updated.

# 5.8 Functions and Routines for Macros

In Program 5.5, %SYSEVALF evaluated the logical comparison &NPTS<1. %SYSEVALF is a general evaluation function. A short list with description of a few useful macro functions are given below:

* %SYSEVALF: general evaluation function
* %EVAL: function for integer arithmetic
* %INDEX: position of a string in a source string
* %LENGTH: length of an argument and
* %SCAN: particular work in an argument
* %SUBSTR: substring extracted from an argument
* %UPCASE: converts argument to all uppercase

Program 5.8 provides a short example to illustrate some of these functions with results from executing this program displayed in Table 5.11.

Program 5.8 Illustrating macro functions with log results

%let summer = June July August;

%let pickmth = 3;

%let mymonth = %scan(&summer, &pickmth); \* pickmth word of summer;

%let mymonth3 =%substr(&summer, 11, 3); \* start @ position 11 and move 3;

%let upper\_month3 = %upcase(&mymonth3);

%put Summer=&summer;

%put Length of '&summer' = %length(&summer);

%put Where is Aug in the '&summer'? = %index(&summer, Aug);

%put Month picked = &pickmth;

%put Which month? = &mymonth;

%put Which month (3 letters)? = &mymonth3;

%put Upper case (3 letters)? = &upper\_month3;

Table 5.11 SAS LOG Illustrating Macro Function

Summer=June July August

Length of '&summer' = 16

Where is Aug in the '&summer'? = 11

Month picked = 3

Which month? = August

Which month (3 letters)? = Aug

Upper case (3 letters)? = AUG

In addition to the macro functions described above, the functions SYMPUT, SYMPUTX, and SYMGET are incredibly useful. The SYMPUT and SYMPUTX functions can be used to assign a SAS variable value to a macro variable. The SYMGET function assigns a macro variable value to a SAS variable.

Worth Noting:

SYMPUT: SAS variable value🡪Macro variable

SYMPUTX: SAS variable value🡪Macro variable (removing leading and trailing blanks)

SYMGET: SAS variable🡨Macro variable value

Program 5.9 provides a program in which macro variables &MBROOD1, &MBROOD2, &MBROOD3, and &MTOTAL are constructed from the variable BROOD and the index variable I.[[1]](#footnote-1) A %DO loop over these variables generates separate plots of the number of young versus concentration for each of the broods.

Program 5.9 SYMPUT function in a macro

**data** nitrofen;

filename cdub URL "http://www.users.miamioh.edu/baileraj/datasets/ch2-dat.txt";

infile cdub firstobs=**16** expandtabs missover pad;

input @**9** animal **2.**

@**17** conc **3.**

@**25** brood1 **2.**

@**33** brood2 **2.**

@**41** brood3 **2.**

@**49** total **2.**;

**run**;

**data** test; set nitrofen;

brood=**1**; conc=conc; nyoung=brood1; output;

brood=**2**; conc=conc; nyoung=brood2; output;

brood=**3**; conc=conc; nyoung=brood3; output;

**run**;

**%macro** ***threeregs***;

proc sort data=test out=test;

by brood;

run;

data \_null\_;

set test;

by brood;

if first.brood then do;

i+**1**;

ii = left(put(i,**2.**));

call symput('mbrood'||ii,trim(left(brood)));

call symput('mtotal',ii);

/\*

Alternative with symputx

call symputx('mbrood'||ii, brood);

call symputx('mtotal',ii);

\*/

end;

run;

%do ibrood = **1** %to &mtotal;

proc sgplot data=test;

where brood=&&mbrood&ibrood;

reg x=conc y=nyoung / degree=**2**;

title "Plot: # Young vs. Nitrofen Conc.";

title2 "[brood &&mbrood&ibrood]";

%put 'ibrood' = &ibrood;

%put '&&mbrood&ibrood' = &&mbrood&ibrood;

run;

%end;

proc sgpanel data=test;

title "SGPANEL alternative display [avoids macros & looping]";

panelby brood / columns=**3**;

reg x=conc y=nyoung / degree=**2**;

run;

**%mend**;

ods rtf file="&Folder\&subFolder\ch5-fig5.13.rtf"

image\_dpi=**300**

style=sasuser.customSapphire;

%***threeregs***()

**quit**;

ODS RTF close;

Figures 5.2 - 5.5 indicate that the impact of nitrofen is not observed in the first brood, but the impact is dramatic in the later broods.

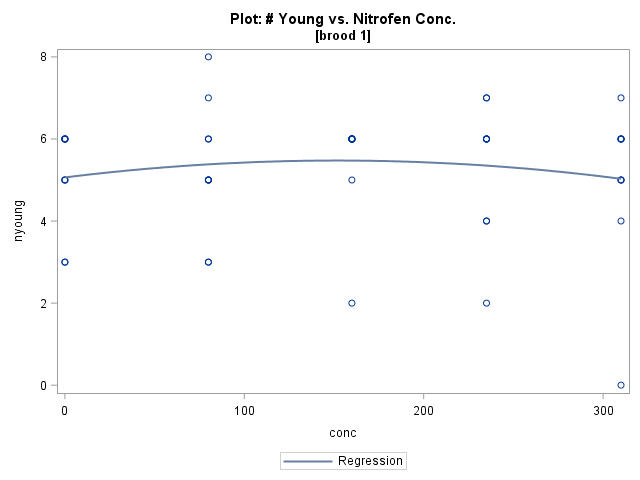


Figure 5.2 Brood 1 count vs. nitrofen concentration

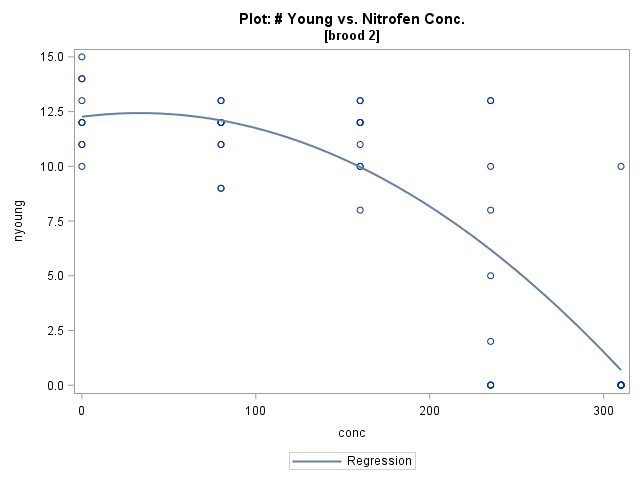


Figure 5.3 Brood 2 count vs. nitrofen concentration

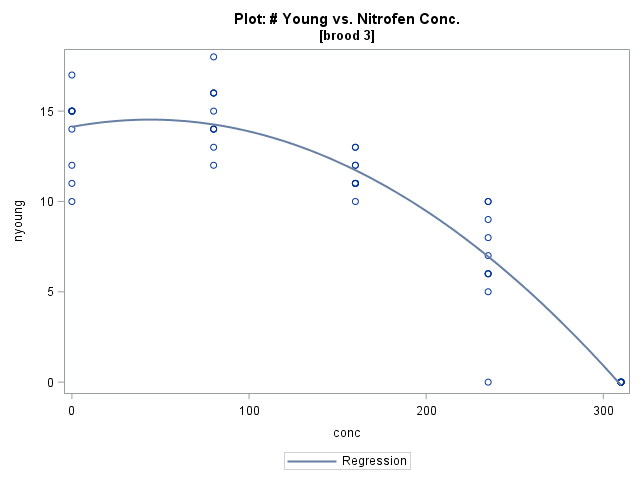


Figure 5.4: Brood 3 count vs. nitrofen concentration

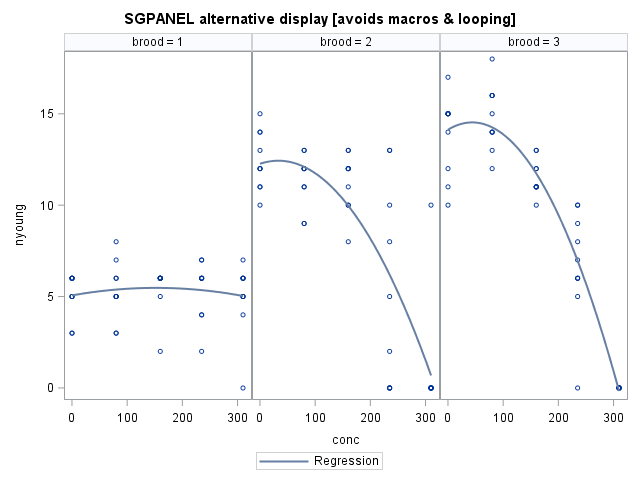


Figure 5.5 SGPANEL plot produced by the %threeregs macro

# 5.9 Case Study: Macro for constructing training and test data set for Model Comparison

We macro-ize our construction of training and test data sets in this case study. (Macro-ize? Macro-size? Macro-size sounds like we are making something bigger.) In Chapter 4 we wrote a program to generate training and test data sets that we then used to train three different regression models and then evaluated performance of the models when predicting responses in the test data that wasn’t used to develop the models. This is a task that you might be interested in doing regularly, and a general solution would be attractive.

Worth Noting: If you find yourself regularly faced with the same problem, then invest the time to solve it once with a general solution. It may take a little longer to produce the general solution; however, you will reap benefits later.

Program 5.10 extracts the pieces of Program 4.17 that generates the training and test samples. In addition, comments are included to highlight where macro variables could be substituted for a hardcoded values.

Program 5.10: Macro for building training / test data sets based on Program 4.19 Selecting a subset of a data set to train a model

**data** train\_test;

set SASHELP.CARS; \* <-------- data set could be specified;

call streaminit(**7525**); \* <-------- Seed could be specified;

retain ntest ntrain **0**;

pick\_test = RAND("uniform");

if (pick\_test < **0.25** and ntest < **107**) then do;

\* <---- fraction could be set, 107 = n\*fraction;

ntest = ntest + **1**;

dsn= "test ";

end;

else if (ntrain < **321**) then do; \* <---- 321 = n\*(1-fraction);

ntrain = ntrain + **1**;

dsn = "train";

end;

**run**;

\* define new variable to be Y for the training data but

missing for the test data - produces prediction for both

data sets - also construct the quadratic and cubic terms;

**data** reg\_train; \* <----- provide a name for the output data set;

set train\_test;

obs\_no = \_N\_; \* <----- useful to have for merging diff model fits;

if dsn="train" then Y\_MPG\_Highway = MPG\_Highway;

\* <--- specify name of the response variable;

else Y\_MPG\_Highway = **.**;

**run**;

\* debugging code to check variable and data set construction;

**proc** **print** data=reg\_train (obs=**4**); \* <------ print out if want to debug;

**run**;

**proc** **freq** data=reg\_train;

table dsn;

**run**;

The first step in implementing a general macro is to replace hard-coded variable values with macro variables. This is done in Program 5.11 and %LET is used to assign values to the macro variables to test the code.

Program 5.11: Macro for buidling training / test data sets - macro variables set

\* start with macro variables that replace hard coded values;

%let dsn = SASHELP.CARS;

%let seed = 7525;

%let test\_frac = 0.25;

%let nobs = 428;

\* number of observations in the CARS data set - need to calculate in future;

%let y\_var = MPG\_HIGHWAY;

%let new\_y\_var = Y\_&y\_var;

%put y\_var new\_y\_var;

%let outdsn = reg\_train;

%put \_USER\_;

\* need to calculate the size of the training and test data sets;

**data** \_NULL\_;

ntest\_size = &nobs \* &test\_frac;

ntrain\_size = &nobs - ntest\_size;

put ntrain\_size ntest\_size;

call symput('Mtrain\_n', ntrain\_size);

call symput('Mtest\_n', ntest\_size);

**run**;

**data** train\_test;

set &dsn; \* <-------- &dsn now used ;

call streaminit(&seed); \* <-------- &seed now specified;

retain ntest ntrain **0**;

pick\_test = RAND("uniform");

if (pick\_test < &test\_frac and ntest < &Mtest\_n) then do;

\* <---- &fraction could be set, 107 = n\*fraction;

ntest = ntest + **1**;

dsn= "test ";

end;

else if (ntrain < &Mtrain\_n) then do;

\* <---- 321 = n\*(1-fraction) = n - n\_test;

ntrain = ntrain + **1**;

dsn = "train";

end;

**run**;

\* define new variable to be Y for the training data but

missing for the test data - produces prediction for both

data sets - also construct the quadratic and cubic terms;

**data** &outdsn;

set train\_test;

obs\_no = \_N\_;

\* <----- useful to have for merging diff model fits;

if dsn="train" then &new\_y\_var = &y\_var;

else &new\_y\_var = **.**;

**run**;

\* debugging code to check variable and data set construction;

**proc** **print** data=&outdsn (obs=**4**); \* <------ print out if want to debug;

**run**;

**proc** **freq** data=&outdsn;

table dsn;

**run**;

The results of running this code (not included since you have already seen this in Ch. 4) confirm that the macro variable substitution works. The SYMPUT function is used to define the macro variables that contain the size of the training and test data sets as we see in the code portion reproduced below:

**data** \_NULL\_;

ntest\_size = &nobs \* &test\_frac;

ntrain\_size = &nobs - ntest\_size;

put ntrain\_size ntest\_size;

call symput('Mtrain\_n', ntrain\_size);

call symput('Mtest\_n', ntest\_size);

**run**;

One remaining challenge with this code is that &nobs, the macro variable containing the number of observations in the input data set &dsn is being specified versus calculated. This makes the code less general. It is more desirable to calculate the number of observations of any input data set as part of the macro. In addition, this could provide an error check of whether the data set exists. Program 5.12 is a small macro that calculates the number of observations in a SAS data set. This is based on an example from the SAS Help documentation for %SYSFUNC – see, I told you in Chapter 1 that navigating help would be a useful skill for SAS programming!

Program 5.12: Calculating number of observations in a SAS data set instead of hard coding value

**%macro** nobs\_in\_dsn(dsn);

/\* macro modified from SAS Help documentation for %SYSFUNC

Example 5: Determining the Number of Variables and Observations in a Data Set

functions employed:

%sysfunc: applies function to macro

open: opens a data set and returns a unique identifier

identifier = 0 if data set does not exist

attrn: return value of numeric attribute for a SAS data set

required argument - data set id that OPEN fcn returns (SAS help)

close: close a data set after used

\*/

%global dset nobs;

%let dset = &dsn;

%let dsid = %sysfunc(open(&dset));

%if (&dsid=**1**) %then %do;

%let nobs = %sysfunc(attrn(&dsid,NOBS));

%let rc = %sysfunc(close(&dsid));

%put Dataset &dsn has &nobs observations;

%put dsid = &dsid;

%end;

%else %put &dsn does not exist. Enter a valid data set name.;

**%mend**;

%***nobs\_in\_dsn***(SASHELP.CARS)

%***nobs\_in\_dsn***(NoSuchDataSet)

The OPEN function returns an identifier that will be zero if the data set does not exist. Given this identifier, the %SYSFUNC macro function applies the ATTRN function to the data set to extract the number of observations and save this to a macro variable. This macro is tested with a data set that exists, SASHELP.CARS, and another that does not, NoSuchDataSet. Table 5.12 is the LOG output produced from executing this macro twice.

Table 5.14 Applying Macro to Extract the Number of Observations in a Data Set

%nobs\_in\_dsn(SASHELP.CARS)

Dataset SASHELP.CARS has 428 observations

dsid = 1

%nobs\_in\_dsn(NoSuchDataSet)

NoSuchDataSet does not exist. Enter a valid data set name.

The final step is to add a macro wrapper around the code with the macro variables and the code to extract the number of observations in a data set. Program 5.15 is the result.

Program 5.15: Macro for building training / test data sets - macro variables set \*/

**%macro** train\_test\_gen(dsn, seed, test\_frac, y\_var, debug, outdsn);

%\* INPUT:

dsn = input data set

seed = starting seed for random number generator

test\_frac = fraction of dsn to be assigned to TEST sample

1-test\_frac assigned to TRAINING sample

y\_var = name of variable in dsn that will be used in later modeling

debug = produce output for debugging checks

OUTPUT:

outdsn = output dataset with new variable

Y\_(y\_var) = value of the (y\_var) for training data and

= . (missing) if test data

obs\_no = observation number (useful for merging different fits)

Note: code generalized after testing

;

%\* need to define new Y variables;

%let new\_y\_var = Y\_&y\_var;

%\* part of macro nobs\_in\_dsn() needed to generate macro variable

with # of observations in the data set;

%global dset nobs;

%let dset = &dsn;

%let dsid = %sysfunc(open(&dset));

%\* check to make sure dataset exists;

%if (&dsid = **0**) %then %put Invalid data set name -> &dsn <- provided;

%else %do; \* <<<<<<<<<<<<<<<<< data set exists;

%let nobs = %sysfunc(attrn(&dsid,NOBS)); \*nobs in data set;

\* need to calculate the size of the training and test data sets;

data \_NULL\_;

ntest\_size = &nobs \* &test\_frac;

ntrain\_size = &nobs - ntest\_size;

call symput('Mtrain\_n', ntrain\_size);

call symput('Mtest\_n', ntest\_size);

run;

data train\_test;

set &dsn;

call streaminit(&seed);

retain ntest ntrain **0**;

pick\_test = RAND("uniform");

if (pick\_test <= &test\_frac) then do;

if (ntest < &Mtest\_n) then do;

ntest = ntest + **1**;

dsn= "test ";

end;

else do; \* test sample filled - put obs into training set;

ntrain = ntrain + **1**;

dsn = "train";

end;

end;

else do;

if (ntrain < &Mtrain\_n) then do;

ntrain = ntrain + **1**;

dsn = "train";

end;

else do; \* training sample filled - put obs into test set;

ntest = ntest + **1**;

dsn= "test ";

end;

end;

run;

data &outdsn;

set train\_test;

obs\_no = \_N\_;

if dsn="train" then &new\_y\_var = &y\_var;

else &new\_y\_var = **.**;

run;

\* debugging code to check variable and data set construction;

%if (&debug=TRUE) %then %do;

%put \_USER\_; \* write out user defined macro variables;

proc print data=&outdsn (obs=**4**);

run;

proc freq data=&outdsn;

table dsn;

run;

%end;

%let rc = %sysfunc(close(&dsid)); \* close the data set;

%end; \* <<<<<<<< if data set found ;

**%mend**; \* of macro train\_test\_gen ;

/\* Now, run the macro with a few test cases \*/

ods rtf file="&Folder\&subFolder\ch5-fig5.15.rtf"

image\_dpi=**300**

style=sasuser.customSapphire bodytitle;

title "Testing train\_test\_gen macro";

\* test the macro to generate test and training values;

%train\_test\_gen(dsn=SASHELP.CARS, seed=7525, test\_frac=0.25,

y\_var=MPG\_HIGHWAY, debug=FALSE, outdsn=CarTrainTest)

\* look at a different data set;

%train\_test\_gen(dsn=SASHELP.COMET, seed=8675309, test\_frac=0.30,

y\_var=LENGTH, debug=TRUE, outdsn=CometTrainTest)

\* change the test sample fraction;

%train\_test\_gen(dsn=SASHELP.COMET, seed=9035768, test\_frac=.10,

y\_var=LENGTH, debug=TRUE, outdsn=C2)

\* how about a data set that doesn't exist?;

%train\_test\_gen(dsn=SASHELP.CUPID, seed=8675309, test\_frac=0.30,

y\_var=LENGTH, debug=TRUE, outdsn=CometTrainTest)

title;

ods rtf close;

Run the test cases here and consider some of your own. Can you break this macro by giving it incorrect input? Alas the answer is ‘yes.’ We build in a check to confirm that the data set exists but we didn’t confirm that the y\_var exists or that we had a valid test fraction. If you are building code for production use, then you may want to add more parameter checks as part of running the code.

# 5.10 Case Study: Processing Multiple Data Sets

A common challenge in statistical programming is processing a large number of data sets. For example, you might have data from individual subjects that are stored in separate files such as spreadsheets. Or, you might have separate data files for different dates. While these examples require processing many files, let’s look at a simple example that captures the essence of the problem.

In this example, the general problem is merging data sets that contain time and temperature data. A different data set is available for each date. The construction of three time and temperature data sets is shown in Program 5.16. The data sets are named AUG03, AUG05, and AUG17, which are three observation dates in the month of August. The temperature (TEMP) variable is assigned a value that increases until a late afternoon maximum, and then decreases from that point onward.

Program 5.16 Constructing three data sets for later use

**data** aug03;

call streaminit(**9035768**);

mydate='AUG03';

do time=**1** to **24**;

temp = **74** - abs(time-**16**) + RAND('Normal',**0**,**0.5**);

output;

end;

**run**;

**data** aug05;

mydate='AUG05';

do time=**1** to **24**;

temp = **78** - abs(time-**16**) + RAND('Normal',**0**,**0.5**);

output;

end;

**run**;

**data** aug17;

mydate='AUG17';

do time=**1** to **24**;

temp = **90** - abs(time-**16**) + RAND('Normal',**0**,**0.5**);

output;

end;

**run**;

/\* DEBUGGING BLOCK TO CHECK DATA SET CONSTRUCTION \*/

**data** tester; set aug03 aug05 aug17;

**proc** **print** data=tester;

id time;

**run**;

**proc** **sgplot** data=tester;

series x=time y=temp / group=mydate;

symbol interpol=join;

**run**;

If you want to read an arbitrary number of arbitrarily named data sets, then you need to make sure of a few things. In Program 5.17, the names of the data sets that you want to merge are read. These data set names are assigned as the values of the macro variables &DSN1, &DSN2, and so on using SYMPUTX. This task is similar to constructing the broods in a previous example. A count of the number of data sets processed is stored as a macro variable.

Program 5.17 Reading data set names and constructing macro variables

**data** \_null\_;

\* read data sets and create macro variable with name of each;

retain counter **0**;

input times $ @@;

counter = counter + **1**;

put times counter;

\* create a macro variable with each data set name;

\* create macro variable name with total number of DSNs;

call symputx('dsn' || LEFT(counter), times);

call symputx('num\_data\_sets', counter);

datalines;

aug03 aug05 aug17

;

**run**;

%put \_user\_;

The values of the user-defined macro variables after executing the code in Program 5.17 are requested by %put \_user\_. The macro variables and values are the following:

GLOBAL NUM\_DATA\_SETS 3

GLOBAL DSN1 aug03

GLOBAL DSN2 aug05

GLOBAL DSN3 aug17

These macro variables are global because they are defined in open code, not defined internally in a macro.

Program 5.18 contains the macro program. The macro program builds a DATA step that concatenates the **&num**\_**data**\_**sets** data sets whose names are the values of the macro variables (for example, **&dsn1**, **&dsn2,** and **&dsn&num**\_**data**\_**sets**). This task occurs when you execute the code in Program 5.17. In Program 5.18, the macro program that names the combined data set is invoked, the combined data set is printed, and a plot of the data is generated.

Program 5.18 Defining and running a macro program to combine data sets

**%macro** concatenator(combine);

data &combine;

set

%do ii = **1** %to &num\_data\_sets;

&&dsn&ii

%end;

;

run;

**%mend** concatenator;

/\*

concatenate the three data sets and print results

\*/

options mprint mlogic;

%***concatenator***(combine=all3)

title;

ods rtf file="&Folder\&subFolder\ch5-fig5.15-16.rtf"

image\_dpi=**300**

style=sasuser.customSapphire;

**proc** **print** data=all3;

**run**;

**proc** **sgplot** data=all3;

series x=time y=temp / group = mydate;

yaxis label="Temperature (deg. F)"

values=(**55** to **95** by **10**);

xaxis label="Time (h) since midnight"

values=(**0** to **24** by **6**);

**run**;

ods rtf close;

Setting the mlogic and mprint options provides a trace of the execution of this macro program. From this trace, you can see that the following code is the result of %concatenator(all3):

data all3;

set aug03 aug05 aug17;

run;

Here are the results of the macro options that were set before the macro was invoked:

MLOGIC(CONCATENATOR): Beginning execution.

MLOGIC(CONCATENATOR): Parameter COMBINE has value all3

MPRINT(CONCATENATOR): data all3;

MLOGIC(CONCATENATOR): %DO loop beginning; index variable

N\_DSN; start value is 1; stop value is 3; by value is 1.

MLOGIC(CONCATENATOR): %DO loop index variable N\_DSN is now 2;

loop will iterate again.

MLOGIC(CONCATENATOR): %DO loop index variable N\_DSN is now 3;

loop will iterate again.

MLOGIC(CONCATENATOR): %DO loop index variable N\_DSN is now 4;

loop will not iterate again.

MPRINT(CONCATENATOR): set aug03 aug05 aug17 ;

MPRINT(CONCATENATOR): run;

NOTE: There were 24 observations read from the data set

WORK.AUG03.

NOTE: There were 24 observations read from the data set

WORK.AUG05.

NOTE: There were 24 observations read from the data set

WORK.AUG17.

NOTE: The data set WORK.ALL3 has 72 observations and 3

variables.

NOTE: DATA statement used (Total process time):

real time 0.01 seconds

cpu time 0.00 seconds

The printout of the combined data set is shown in Table 5.13.

Figure 5.13 Printout of the combined data set (Edited)

| **Obs** | **mydate** | **time** | **temp** |
| --- | --- | --- | --- |
| **1** | AUG03 | 1 | 58.8312 |
| **2** | AUG03 | 2 | 59.7219 |
|  | … | … | … |
| **24** | AUG03 | 24 | 66.7752 |
| **25** | AUG05 | 1 | 63.4003 |
| **26** | AUG05 | 2 | 64.0923 |
|  |  | … | … |
| **48** | AUG05 | 24 | 69.1808 |
| **49** | AUG17 | 1 | 74.7468 |
| **50** | AUG17 | 2 | 76.0121 |
|  |  | … | … |
| **71** | AUG17 | 23 | 83.6768 |
| **72** | AUG17 | 24 | 82.3141 |

The plot of the data for the three dates is shown in Figure 5.6. A separate time and temperature profile is displayed for each of the three dates.



Figure 5.6 Plot of the temperature-hour profile on the three dates

# 5.10 References [to be moved to reference list at the end of the book?]

Burden, Richard L., and J. Douglas Faires. 1989. *Numerical Analysis*. 4th Ed. Boston: PWS-KENT Publishing Co.

Burlew, Michele M. 2014. *SAS Macro Programming Made Easy*. 3rd Ed. SAS Institute Inc. Cary, NC.

Carpenter, Art. 2016. *Carpenter’s Complete Guide to the SAS Macro Language*. 3rd Ed. Cary, NC: SAS Institute Inc.

SAS Institute Inc. 2014. SAS Certification Prep Guide: Advanced Programming for SAS®9. 4th Ed. Cary, NC: SAS Institute Inc.

# 5.11 Exercises

1. Nest the simulation of the two-group *t*-test from Chapter 3, “Case Study 1: Is the Two-Sample t-Test Robust Enough for Heterogeneous Variances?” in a macro. (The code is included in this exercise.) The parameters should include a seed (MYSEED), the sample sizes (N\_X and N\_Y), the population means (Mu\_X and Mu\_Y), the population sigmas (SIG\_X and SIG\_Y), and the number of simulation (Nsims).

Run your macro to estimate the Type I error rates for combinations of N\_X and N\_Y=[5, 15, 25]x[5, 15, 25] and SIG\_Y/SIG\_X=1, 2, 4. Estimate the power for the same conditions with Mu\_X–Mu\_Y=1. You can use macro variables to build this code.

/\* Problem: Explore whether t-test really is robust to violations of the equal variance assumption

Strategy: See if the t-test operates at the nominal Type I error rate when the unequal variance assumption is violated

\*/

**data** twogroup;

array x{**10**} x1-x10;

array y{**10**} y1-y10;

call streaminit(**11223344**);

do isim = **1** to **10000**;

/\* generate samples X~N(0,1) Y~N(0,4) - normal case \*/

do isample = **1** to **10**;

x{isample} = RAND('normal',**0**,**1**);

y{isample} = RAND('normal',**0**,**2**);

end;

/\* calculate the t-statistic \*/

xbar = mean(of x1-x10);

ybar = mean(of y1-y10);

xvar = var(of x1-x10);

yvar = var(of y1-y10);

s2p = (**9**\*xvar + **9**\*yvar)/**18**;

tstat = (xbar-ybar)/sqrt(s2p\*(**2**/**10**));

Pvalue = **2**\*(**1**-probt(abs(tstat),**18**));

Reject05 = (Pvalue <= **0.05**);

keep xbar ybar xvar yvar s2p tstat Pvalue Reject05;

output;

end; \* end of the simulation loop;

**run**;

/\*

proc print;

run;

\*/

**proc** **freq**;

table Reject05;

**run**;

1. Nest the simulation of the Monte Carlo estimate of P(0<Z<1.645), “Case Study 2: Monte Carlo Integration to Estimate an Integral,” in a macro. Macro variables should be defined for limits of integration, number of simulated data points, and optional construction of graphical displays.
2. Write a macro that calculates the percentile-based bootstrap CI for the mean response of a specified variable in a specified data set at a specified confidence coefficient. For more challenge, also implement a biased-corrected and accelerated bootstrap and set a macro parameter to determine which method is generated.
3. Write a macro to fit a multiple regression model where Y is identified as one macro variable, and each X is identified as another macro variable. Your macro should do the following:
4. Fit the model.
5. Plot the residuals versus each X.
6. Plot the residuals versus Y.
7. Create a normal probability plot of the residuals.

The macro parameters should include the data set name, the response variable, the predictor variables, and a title.

Make sure that an argument is passed for the data set, response variable, and predictor variables. If an argument is not passed, generate an error message.

Test your macro with the SASHELP.FISH data. Fit a multiple regression model predicting the Weight as a function of Width and Length1.

If you really want to go crazy with this exercise, have the macro do the following:

1. Plot the residuals versus each X squared.

Plot the residuals versus XiXj (pairs of the predictor variables).

1. Construct a macro to calculate upper tail probabilities for standard statistical distributions (Z, T, Chi-square and F). This macro should include checks of input arguments (distribution must by Z, T, Chi-square or F; appropriate DF must be specified)

1. Nitrofen data set from Bailer, A. John., and James T. Oris. 1994. “Assessing Toxicity of Pollutants in Aquatic   
    Systems.” Chapter 2 in *Case Studies in Biometry*. Nicholas Lange, [Louise Ryan](http://www.amazon.com/s/ref=ntt_athr_dp_sr_2?_encoding=UTF8&sort=relevancerank&search-alias=books&field-author=Louise%20Ryan), [Lynne Billard](http://www.amazon.com/s/ref=ntt_athr_dp_sr_3?_encoding=UTF8&sort=relevancerank&search-alias=books&field-author=Lynne%20Billard), [David   
    Brillinger](http://www.amazon.com/s/ref=ntt_athr_dp_sr_4?_encoding=UTF8&sort=relevancerank&search-alias=books&field-author=David%20Brillinger), [Loveday Conquest](http://www.amazon.com/s/ref=ntt_athr_dp_sr_5?_encoding=UTF8&sort=relevancerank&search-alias=books&field-author=Loveday%20Conquest), [Joel Greenhouse](http://www.amazon.com/s/ref=ntt_athr_dp_sr_6?_encoding=UTF8&sort=relevancerank&search-alias=books&field-author=Joel%20Greenhouse), eds. New Jersey: John Wiley & Sons, Inc. Reprinted with   
    permission of John Wiley & Sons, Inc. [↑](#footnote-ref-1)