SAS handbook

Colton Gearhart

January 30, 2019

Contents

1 Read in data

1.1 Input data

Datalines:

- Used to read data that you enter directly in the program.
- The DATALINES statement is used with an INPUT statement.
- Can add an INFILE statement if need additional SAS instructions to read the desired data.
- Must be the last statement in the DATA step and immediately precedes the first data line.
- Use a null statement (a single semicolon) to indicate the end of the input data.

```
/* example for datalines */
/* read in data */
data work.student;
  infile datalines dlm='?';
  input Name :$10. Sex $1.;
  datalines;
Alfred?M
Alice?F
Barbara?F
;
run;
```

Obs	Name	Sex
1	Alfred	M
2	Alice	F
3	Barbara	F

Infile:

- Used to read data from an external source.
- Files can be txt, csv, etc. (may need extra options depending on source type).
- Can also use a FILENAME statement (outside of the DATA step) to specify data file.

```
/* examples for infile */
/* read in data (txt file) */
data work.student;
  infile '/folders/myfolders/Handbook/Data/Student data.txt';
```

```
  input Name $ Sex $ Age Height Weight;
run;

/* read in data (csv file) */
data work.student;
  infile '/folders/myfolders/Handbook/Data/Student data.csv' firstobs=2 dsd;
  input Name $ Sex $ Age Height Weight;
run;

/* filename */
filename student '/folders/myfolders/Handbook/Data/Student data.txt';

/* read in data (text file) */
data work.student;
  infile student;
  input Name $ Sex $ Age Height Weight;
run;
```

Obs	Name	Sex	Age	Height	Weight
1	Alfred	M	14	69.0	112.5
2	Alice	F	13	56.5	84.0
3	Barbara	F	13	65.3	98.0

* Output for proc import as well.

Proc import:

- Another way to read to read data from an external source.
- Can be used for most file types, but it could result in data errors.
- Note -> xlsx files cannot be read in using an INFILE statement, so have to us PROC IMPORT.

```
/* example for proc import */
/* read in data (xlsx file) */
proc import datafile='/folders/myfolders/Handbook/Data/Student data.xlsx'
    out=work.student dbms=xlsx replace;
  sheet=Enrolled;
  getnames=yes;
run;
```

Types of input styles:

- Modified list input
  - Allows you to read list input with nonstandard data by using SAS informats.
  - Uses a scanning method for locating data values.

  - Data values must be separated by at least one blank (or other defined delimiter).
  - Do not have to specify the location of the data fields.
- Formatted input
  - Enables you to read nonstandard data values that are aligned in columns in the data records.
  - Typically used with pointer controls that enable you to control the position of the input pointer in the input buffer when you read data.

1.2 Options

Here are some common options used with INFILE statement...

DLM=*'list-of-delimiting-characters'*:

- Specifies an alternate delimiter.

DSD (delimiter-sensitive data):

- Use when you want to treat two consecutive delimiters as a missing value.
- Also specifies that when data values are enclosed in quotation marks, delimiters within the value are treated as character data.
- Use the tilde (~) format modifier to retain the quotation marks.

Firstobs=*record number*:

- Specifies the first record for SAS to begin reading the input data records.

Obs=*record number*:

- Specifies the last record for SAS to end reading the input data records.

Missover:

- Use if the last field or fields might be missing and you want SAS to assign missing values to the corresponding variable.

N=*available-lines*:

- Specifies the number of lines that are available to the input pointer at one time.
- When using # pointer controls in an INPUT statement, include a # pointer control that equals the value of the N= option (even if no data is read from that record).

1.3 Modifiers

& (ampersand):

- Enables you to read character values that contain embedded blanks.
- SAS reads until it encounters two consecutive blanks or the defined length of the variable, whichever comes first.

```
/* example for & */
/* read in data */
data work.student;
  input Name &$20. Sex $1.;
  datalines;
Alfred Wise  M
Alice McCarty  F
Barbara Hamilton  F
;
run;
```

Obs	Name	Sex
1	Alfred Wise	M
2	Alice McCarty	F
3	Barbara Hamilton	F

: (colon):

- Enables you to specify an informat, whether character or numeric.
- SAS reads the defined length of the variable or until it encounters a blank column, whichever comes first.

```
/* example for : */
/* read in data */
data work.student;
  input Name :$15. Sex $1.;
  datalines;
Alfred M
AliceInWonderland F
Barbara F
;
run;
```

Obs	Name	Sex
1	Alfred	M
2	AliceInWonderla	F
3	Barbara	F

@ (trailing @):

- Holds an input record for the execution of the next INPUT statement **within the same iteration** of the DATA step.
- Use when you want to read multiple records from the same line of data.
- Can add an indicator (counter) variable for each record.
- The trailing @ must be the last item in the INPUT statement.

```
/* example for @ */
/* read in data */
data work.student;
  /* load Name into input buffer (name gets repeated for each test score) */
  input Name $ @;

  /* loop to add Score to input buffer (and index the test number); then
    output */
  do Test=1 to 4;
    input Score @;
    output;
  end;
  datalines;
Sarah 89 70 90 91
Eric 87 79 90 88
;
run;
```

Obs	Name	Test	Score
1	Sarah	1	89
2	Sarah	2	70
3	Sarah	3	90
4	Sarah	4	91
5	Eric	1	87
6	Eric	2	79
7	Eric	3	90
8	Eric	4	88

@@ (double trailing @):

- Holds the input record for the execution of the next INPUT statement **across iterations** of the DATA step (i.e. there is an implicit output once all the input variables have been read, so @@ holds the record between multiple outputs).
- Use when have more than one record per line within raw data file.
- The double trailing @ must be the last item in the INPUT statement.

```
/* example for @@ */
/* read in data */
data work.student;
  /* hold record in input buffer for the next iteration */
  input Name $ Score1-Score4 @@;
  datalines;
Sarah 89 70 90 91 Eric 87 79 90 88
;
run;
```

Obs	Name	Score1	Score2	Score3	Score4
1	Sarah	89	70	90	91
2	Eric	87	79	90	88

@n and +n (column pointer controls):

- @n moves the pointer to column n (use when columns have nonstandard lengths).
- +n moves the pointer n columns (use when columns have standard lengths).
- Note -> n can also be a numeric variable, expression, character string, etc.

```
/* example for @n */
/* read in data */
data work.student;
  input @13 LastName $12. @25 Sex $1.;
  datalines;
Alfred       Wise         M
Alice        McCarty      F
Barbara      Hamilton     F
;
run;

/* example for +n */
/* read in data */
data work.student;
  input @9 Zip 5. +2 State $2.;
  datalines;
Alfred   41092  GA
Alice    50293  NY
Barbara  70932  OH
;
run;
```

```
/* example for #n */
/* read in data */
data work.student;
  input Name $10. #3 Score2;
  datalines;
```

```
Alfred
78
90
Alice
96
92
Barbara
88
86
;
run;
```

@n

Obs	LastName	Sex
1	Wise	M
2	McCarty	F
3	Hamilton	F

+n

Obs	Zip	State
1	41092	GA
2	50293	NY
3	70932	OH

#n (line pointer controls):

- #n moves the pointer to record n (i.e. jumps to nth line in the raw data for the current record).
- Use when data for a single record is on multiple lines.
- Note -> n can also be a numeric variable or expression.

```
/* example for #n */
/* read in data */
data work.student;
  input Name $10. #3 Score2;
  datalines;
Alfred
78
90
Alice
96
92
Barbara
88
86
;
run;
```

Obs	Name	Score2
1	Alfred	90
2	Alice	92
3	Barbara	86

? and ?? (error reporting modifiers):

- ? suppresses printing the invalid data note when SAS encounters invalid data values.
- ?? suppresses printing the messages and the input lines when SAS encounters invalid data values.

1.4 Informats

- Describe the data value and tells SAS how to convert it.
- Can be specified in the INPUT statement or in an INFORMAT statement (using the INFORMAT statement tells SAS to use the informat in any subsequent input statements).

```
/* example for informats */
/* read in data */
data work.student;
  input Name :$15. Income comma6. Birthday monyy5.;
  datalines;
Alfred 6,000 Apr95
Alice 5,590 Dec96
Barbara 9,200 Feb96
;
run;
```

Obs	Name	Income	Birthday
1	Alfred	6000	12874
2	Alice	5590	13484
3	Barbara	9200	13180

2 Data step stuff

2.1 Drop, keep and rename variables

Dropping, keeping and renaming variables:

- Can do these by using a statement, dataset option or both.

- Results depend on if you specify the data set options on an input or an output data set.
- Order of application in SAS -> For input datasets, drop= and keep= options, then rename= option, then drop and keep statements, then rename statement, then options again for output datasets.
- Statements:
  - Apply to output data sets only.
  - Affect all output data sets.
  - Can be used in DATA steps only.
  - Can appear anywhere in DATA steps (although rename must be after keep / drop).
- Dataset options:
  - Apply to output or input data sets.
  - Affect individual data sets.
  - Can be used in DATA steps and PROC steps.
  - Must immediately follow the name of each data set to which they apply.
- Shorthand notation -> Can use numbered range lists (i.e. x1,x2,...,xn = x1-xn).

```
/* general syntax for dropping, keeping and renaming variables */
/* -> when renaming, have to use one equal sign for each pair */
data work.temp2 (rename=(<oldvars>=<newvars>) keep=<keepvars> drop=<dropvars>
    );
  set work.temp;
  keep <keepvars>;
  drop <dropvars>;
  rename <oldvars=newvars>;
run;
```

```
/* example for dropping, keeping and renaming variables */
/* read in data and modify resulting dataset */
data work.student (rename=(Name=FirstName) drop=Height);
  keep Name Sex Age Height;
  rename Sex=Gender;
  infile '/folders/myfolders/Handbook/Data/Student data.txt';
  input Name $ Sex $ Age Height Weight;
run;
```

Obs	FirstName	Gender	Age
1	Alfred	M	14
2	Alice	F	13
3	Barbara	F	13

2.2 Formats and labels

Formats:

- Change the appearance of a variable?s value in a report.
- The values stored in the data set are not changed.
- Remove a format by specifying a format statement (with desired variables) after the input dataset.

Labels:

- Change the appearance of a variable's name in a report.
- Have to add a label option in PROC PRINT to apply them.

```
/* example for formats and labels
/* read in data and modify resulting dataset */
data work.student;
  label Name='Name of student';
  format Income dollar6. Birthday date5.;
  input Name :$15. Income comma6. Birthday monyy5.;
  datalines;
Alfred 6,000 Apr95
Alice 5,590 Dec96
Barbara 9,200 Feb96
;
run;

/* view dataset */
proc print data=work.student label;
run;
```

Obs	Name of student	Income	Birthday
1	Alfred	$6,000	01APR
2	Alice	$5,590	01DEC
3	Barbara	$9,200	01FEB

```
/* example for removing formats and labels */
/* create new dataset with previous formats removed */
data work.student2;
```

```
  set work.student;
  format Income Birthday;
run;

/* view dataset and don't show labels */
proc print data=work.student2;
run;
```

Obs	Name	Income	Birthday
1	Alfred	6000	12874
2	Alice	5590	13484
3	Barbara	9200	13180

2.3 Conditional logic statements

```
/* general syntax for conditional logic statements */
data work.temp2;
  set work.temp;

  if <condition(s)> then
    <action>;
  else if <condition(s)> then
    <action>;
  else
    <action>;

  if <conditions(s)> then
    do;
      <actions>;
    end;
  else
    do;
      <actions>;
    end;
run;
```

```
/* example for conditional logic statements */
/* read in data and modify resulting dataset */
data work.student;
  length HeightStatus $8.;
  infile '/folders/myfolders/Handbook/Data/Student data.txt';
  input Name $ Sex $ Age Height Weight;

  /* create new variable based on a condition */
  if Height lt 55 then
    HeightStatus='Short';
  else if Height lt 65 then
```

```
    HeightStatus='Normal';
  else
    HeightStatus='Tall';

  /* create new variable based on two conditions using logical operators */
  if Sex eq'M' and Age lt 14 then
    Class='A';
  else if Sex eq'M' and Age ge 14 then
    Class='C';

  if Sex eq 'F' and Age lt 14 then
    Class='B';
  else if Sex eq 'F' and Age ge 14 then
    Class='D';

  /* create same variable using nested if statements */
  if Sex eq 'M' then
    do;

      if Age lt 14 then
        Class='A';
      else
        Class='C';
    end;

  if Sex eq 'F' then
    do;

      if Age lt 14 then
        Class='B';
      else
        Class='D';
    end;
run;
```

Obs	HeightStatus	Name	Sex	Age	Height	Weight	Class
1	Tall	Alfred	M	14	69.0	112.5	C
2	Normal	Alice	F	13	56.5	84.0	B
3	Tall	Barbara	F	13	65.3	98.0	B

2.4 Do loops

```
/* general syntax for do loops */
data work.temp2;
  set work.temp;

  /* do to loop */
  do <index>=<startvalue> to <endvalue> (by <increment>);
```

```
    <actions>;
  end;

  /* do until loop */
  do until <expression>;
    <actions>;
  end;

  /* do while loop */
  do while <expression>
    <actions>;
  end;
run;
```

2.5 Retain statement

- Retains the value of the variable across iterations of the data step.
- Initializes the retained variable to missing or a specified initial value before the first iteration of the data step.
- Use when you want to create an accumulating variable (or a counter variable).
  - If accumulating a variable and one observation has a missing value, the rest of the observations for the accumulating variable will be missing.
  - So have to use SUM function.
- Using the shorthand notation *accumvar+varinterest*, the accumulating variable is initialized to zero and automatically retained.

```
/* general syntax for retaining and accumulating variables */
data work.temp2;
  set work.temp;
  retain <countervar> <accumvar>;

  /* counter variable */
  <countervar>+1;

  /* accumulating variable */
  <accumvar>=<accumvar>+<varinterest>;
  <accumvar>=sum(<varinterest>);
  <accumvar>+<varinterest>;
run;
```

```
/* example for retaining and accumulating variables */
/* read in data */
data work.student;
  label Name='Name of student';
  format Income dollar6. Birthday date5.;
```

```
  input Name :$15. Income comma6. Birthday monyy5.;
  datalines;
Alfred 6,000 Apr95
Alice 5,590 Dec96
Barbara 9,200 Feb96
;
run;

/* modify dataset */
data work.student2;
  set work.student;
  retain TotalIncome;
  TotalIncome+Income;
run;
```

Orginal data

Obs	Name	Income	Birthday
1	Alfred	$6,000	01APR
2	Alice	$5,590	01DEC
3	Barbara	$9,200	01FEB

Modified data

Obs	Name	Income	Birthday	TotalIncome
1	Alfred	$6,000	01APR	6000
2	Alice	$5,590	01DEC	11590
3	Barbara	$9,200	01FEB	20790

2.6 Arrays

- Temporary grouping of SAS variables that are arranged in a particular order.
- Identified by an array name.
- Must contain all numeric or all character variables.
- Exists only for the duration of the current DATA step (is not a variable).
- If variables associated with an array do not exist, SAS creates them.

Tips:

- Helpful when systematically recoding data in many variables (e.g. missing data).
- When the initial value list is specified, all elements behave as if they were named in a RETAIN statement (i.e. this creates a lookup table).

- You can use the keyword _TEMPORARY_ in an ARRAY statement to indicate that the elements are not needed in the output data set (i.e. this creates a temporary lookup table).
- Use * to let SAS figure out how many variables are in the array.
- Of *arrayname*{*} as an argument to a function passes the entire array as if it were a variable list.
- Quick way to do specify all variables of a certain type -> _NUMERIC_ or _CHARATER_ (these automatic variables only refer to variables that have already been defined above it).
- The DIM function returns the number of elements in an array (often used as the stop value in a DO loop).

```
/* general syntax for defining and using arrays */
data work.temp2;
  set work.temp;

  /* define numeric array */
  array <arrayname>{dim} <array elements> <(initial value list)>;
  /* define character array */
  array <arrayname>{dim} <$> <length> <array elements> <(initial value list)>
    ;

  /* loop through each item in array */
  do i=1 to dim(<arrayname>);
    <actions involving array{i}>;
  end;

  /* use array elements as an argument to a function */
  <newvar>=<function>(of <arrayname>{*});
run;
```

```
/* example for defining and using arrays */
/* read in data */
data work.student;
  infile datalines dlm=',' dsd;
  input Name :$15. Income1-Income5;
  datalines;
Alfred,6000,5200,,6750,7900
Alice,5590,8000,7130,,6800
Barbara,,9200,7600,8090,7450
;
run;

/* modify dataset */
data work.student2;
  set work.student;
```

```
  array Income{*} Income1-Income5;

  /* recode missing values */
  do i=1 to dim(Income);
    if Income{i}=. then
      Income{i}=0;
  end;
  drop i;

  /* calculate new variable based off variables in array */
  AvgIncome=mean(of Income{*});
run;
```

Orginal data

Obs	Name	Income1	Income2	Income3	Income4	Income5
1	Alfred	6000	5200	.	6750	7900
2	Alice	5590	8000	7130	.	6800
3	Barbara	.	9200	7600	8090	7450

Modified data

Obs	Name	Income1	Income2	Income3	Income4	Income5	AvgIncome
1	Alfred	6000	5200	0	6750	7900	5170
2	Alice	5590	8000	7130	0	6800	5504
3	Barbara	0	9200	7600	8090	7450	6468

3 Combining data

3.1 Concatenating

- Uses a SET statement to concatenate datasets one after the other.
- If datasets have different variables, SAS will create missing values in the uncommon columns for all of the rows.

```
/* general syntax for concatening datasets */
data work.tempC;
  set work.temp1 work.temp2 ...;
run;
```

3.2 Interleaving

- Arranges rows by the values of the BY variable, by the order of the datasets in which they occur.
- Uses a SET statement with a BY statement.
- Datasets must be sorted by the BY variable.

```
/* general syntax for interleaving datasets */
/* -> each dataset must be sorted by <byvar> */
data work.tempC;
  set work.temp1 work.temp2 ...;
  by <byvar>;
run;
```

3.3 Combining rows (one-to-one)

- Combines the first observation from all the datasets into the first observation of the new dataset, the second row in all datasets into the second row in the new dataset, and so on.
- Uses several SET statements.
- If datasets have the same variables, the values that are read in from the last dataset will replace the values that are read in from earlier datasets.

```
/* general syntax for combining rows (one-to-one) */
data work.tempC;
  set work.temp1 work.temp2;
  set work.temp2;
  ... ...;
run;
```

3.4 Match merging

Overall -> All three types of match merging:

- Combine datasets based on a common variable (could be multiple).
- Use a MERGE statement.
- Two or more datasets must be listed in the MERGE statement.
- Variables in the BY statement must be common to all datasets.
- Datasets must be sorted by the variables listed in the BY statement.

- The order of datasets in the MERGE statement doesn't impact results, just the order of the variables in the new dataset will differ.

One-to-one:

- Use when a single observation in one dataset is related to exactly one observation in another dataset based on the BY values (i.e. BY values have only one match in the other datasets).

One-to-many:

- Use when a single observation in one dataset is related to more than one observation in another dataset based on the BY values (i.e. BY values have multiple matches in the other datasets).

```
/* general syntax for match merging datasets (one-to-one and one-to-many) */
/* -> each dataset must be sorted by <byvar> */
data work.tempC;
  merge work.temp1 work.temp2 ...;
  by <byvar>;
run;
```

Non-matches:

- Use when at least one observation in one dataset is unrelated to any observation in another dataset based on the BY values (i.e. BY values do not have to have a match in the other datasets).
- The in= dataset option creates a temporary numeric variable that indicates whether that dataset contributed to building the current observation of the new dataset.
  - *invar*=0 if that dataset did not contribute and 1 if it did.
  - Helpful when you want to perform some action based on which dataset an observation came from (e.g. subsetting new dataset to only observations from certain datasets; or keeping only records with a match).
  - *invar* is only available during the DATA step in which it occurs -> So if it is needed for later DATA steps, it needs to be assigned to another variable.

```
/* general syntax for match merging datasets with non-matches */
/* -> each dataset must be sorted by <byvar> */
data work.tempC;
  merge work.temp1 (in=<invar1>) work.temp2 (in=invar2) ...;
  by <byvar>;

  /* assign in variables for later use */
  dataset1=<invar1>;
  dataset2=<invar2>;
  ...;
```

```
11
12   /* subset combined dataset based on which dataset contributed */
13   if dataset1=1;
14
15   /* keep only records with a match */
16   if dataset1 or dataset2 or ...;
17 run;
```

4 Export and write

4.1 Datasets

```
1 /* general syntax for exporting a dataset with proc export (to a csv file) */
2 /* -> replace overwrites the existing file */
3 /* -> label writes variable labels as first row instead of variable names */
4 proc export data=work.temp
5     outfile=<filename (with path)>
6     dbms=csv replace <(label)>;
7 run;
```

```
1 /* general syntax for exporting a dataset with an ods statement (to a csv
      file) */
2 ods csv file=<filename (with path)>;
3
4 /* close file */
5 ods csv close;
```

4.2 Output

```
1 /* general syntax for writing output */
2 /* -> <filetype> -> use rtf for a word file and pdf for a pdf file
3 /* -> bodytitle option is only for rtf
4 /* open output */
5 ods <filetype> file=<filename (with path)>.<filetype>
6   <bodytitle> style=journal;
7
8 /* select only a certain output items from a procedure */
9 ods select <tablename>;
10 /* customize title for a procedure's output */
11 title <title (string)>;
12
13 /* run procedure that generates output */
14
15 /* ... */
16
17 /* close output */
18 ods <filetype> close;
```

4.3 Reports with put statement

- _NULL_ dataset allows for a DATA step without storing the dataset.
- PUT statement writes to the SAS log.
- If FILE statement is specified, then PUT writes to the specified location.
- MOD option in the FILE statement appends text to the end of the file.
- Do not need to put $ to indicate character variables and can also specify formats after variables.
- Can use column pointer controls to organize report.
- Can use multiple put statements (e.g. if want to add a line of string constants).
- To make an overall header for the report, you can use two DATA steps:
  - The first only has the FILE statement and the PUT statements that write the header.
  - Then the second on actually has the dataset and the FILE statement with the MOD option so that you can append the actual report.
- If want to organize report by a BY variable:
  - Need to sort the data by the BY variable and add a BY statement to the DATA step.
  - Then can use if first.*byvar* and add a header for the BY group and if last.*byvar* to separate groups.

```
/* general syntax for reports with put statement */
/* / in put statement is a line break */
data _null_;
  set work.temp;
  file <filename (with path)> <(mod)>;

  /* write data values */
  put <vars> <(formats)>;

  /* write a blank line */
  put;

  /* write string constants */
  put <some text> / <some more text >;
run;
```

```
/* example for reports with put statement */
/* read in data */
```

```
data work.student;
  infile '/folders/myfolders/Handbook/Data/Student data.txt';
  input Name $ Sex $ Age Height Weight;
run;

/* sort dataset */
proc sort data=work.student;
  by Sex;
run;

/* create header for report */
data _null_;
  file '/folders/myfolders/Handbook/Output/Report - Put.txt';
  put 'Information about students';
  put '---------------------------';
  put;
run;

/* actually create report */
data _null_;
  set work.student;
  by Sex;

  /* append to file with header */
  file '/folders/myfolders/Handbook/Output/Report - Put.txt' mod;

  /* write header with information for the first observation of each sex */
  if first.Sex then
    do;
      put 'Gender = ' Sex;
      put;
      put @1 'Name' @15 'Age' @20 'Height' @27 'Weight' / @1 '----' @15
        '---' @20 '------' @27 '------';
    end;

  /* write out the data for all records */
  put @1 Name @15 Age @20 Height @27 Weight;

  /* separate with blank line */
  if last.Sex then put;
run;
```

```
Information about students
--------------------------

Gender = F

Name          Age  Height Weight
----          ---  ------ ------
Alice         13   56.5   84
Barbara       13   65.3   98
Carol         14   62.8   102.5
Jane          12   59.8   84.5
Janet         15   62.5   112.5
Joyce         11   51.3   50.5
Judy          14   64.3   90
Louise        12   56.3   77
Mary          15   66.5   112

Gender = M

Name          Age  Height Weight
----          ---  ------ ------
Alfred        14   69     112.5
Henry         14   63.5   102.5
James         12   57.3   83
Jeffrey       13   62.5   84
John          12   59     99.5
Philip        16   72     150
Robert        12   64.8   128
Ronald        15   67     133
Thomas        11   57.5   85
William       15   66.5   112
```

5 Proc procedures

Here are notes about some PROC procedures...

5.1 Proc sql

SQL (Structure Query Language):

- Can sort, summarize, subset, join (merge), concatenate datasets, create new variables, print results or create a new table all in a single step.
- Terminology (SAS <–> SQL equivalent):
  - Dataset <–> Table.
  - Observation <–> Row.
  - Variable <–> Column.

- Can add multiple CREATE TABLE, SELECT, etc. statements within a single PROC SQL step.
- The SQL procedure is finished after the QUIT statement.

```
/* general syntax / flow for proc sql */
/* create dataset */
proc sql;
  create table work.temp2 as
    select <vars>
    from work.temp
    where <condition>
    group by <aggvar>
    order by <sortvar>;
quit;

/* specific syntaxes for proc sql */
proc sql;

  /* create a new sas dataset */
  create table work.temp2 as ...

  /* select all variables from the old table */
  select * from work.temp

  /* rename variables */
  select <old var name> as <new var name>

  /* calculate new variable */
  /* -> can also use other functions such as max(), avg(), etc. */
  select <oldvar with additional calculations> as <newvar>

  /* conditionally calculate new variable */
  /* -> there is no if/then statement, so have to use case/when */
  /* -> without as <newvar>, the calculated variable just has a blank header
    */
  select
    case
      when <condition using old value> then <newvar value>
      when <condition using old value> then <newvar value>
      ...
      else <newvar value>
    end as <newvar>

  /* format and label variable */
  select <oldvar> format=<sas format> label=<label>

  /* subset dataset */
  where <expression>

  /* remove duplicates */
```

```
46   select distinct <var>
47
48   /* concatenate tables (by default, this keeps only unique observations) */
49   select * from work.temp1 union select * from work.temp2
50
51   /* join tables */
52   /* -> aliases are used to identify which table a variable came from */
53   select <alias1>.<old var name> <alias2>.<old var name>
54     from <old table1> <alias1> <inner, left, right, etc.> join <old table2> <
     alias2>
55       on (<alias1>.<key variable1>=<alias2>.<key variable2>);
56 quit;
```

5.2 Proc report

Options:

- HEADLINE adds a line after the column headings.
- HEADSKIP option adds a blank line under the column heading (or line from above option).

TITLE statement:

- Used to specify the title at the top of each page.

COLUMN statement:

- Used to list each report column.
- By default, column headings are their respective labels, not the variable names.

DEFINE statement:

- Each column then has a DEFINE statement that describes how that column is created and formatted (via slash / options).
- Define types:
  - The word directly after the slash specifies the define type for that column.
  - DISPLAY:
    - Simply displays the column.
  - ORDER:
    - Specifies the column used to sort the report.
  - GROUP:

    * Aggregates all the observations with the same unique combination of grouped variables.
    * Not very useful unless it is used with the ANALYSIS define type.
    * You can specify the order of the rows within the group by using the ORDER= option of the DEFINE statement.
- ANALYSIS:
    * Lets you specify for that column any of the statistics used in PROC MEANS, SUMMARY and UNIVARIATE.
    * The statistics are calculated for the group you defined.
    * If you want to calculate more than one statistic on the same column, you create an alias in the COLUMN statement (*column=columnalias*).
- ACROSS:
    * Use when you want a report where all the unique values of a variable have their own column.
    * This is the easiest way to create cross-tab.
    * Can specify a variable to be nested under the across variable by separating the across variable and the nested variable with a comma in the COLUMN statement.
    * If you use dashes (or some other character) as the first and last characters in the ACROSS column header, they span all the columns.
- COMPUTED:
    * Used to compute your own values from the other data in your report.
    * Besides the DEFINE statement for each computed column, you need to write a COMPUTE block which starts with a COMPUTE statement and ends with ENDCOMPUTE.
    * A COMPUTE block can contain, more or less, everything allowed in a DATA step including macro variables and %INCLUDE (can also reference any report item from within the block).
    * Can use the automatically defined _C#_ variables to make it easier (e.g. sum(_C2_, _C3_, _C5_) is the sum of the second, third and fifth columns).
    * If don't know how many columns ACROSS define type will create, just use a really big _C#_ (extra variables won't hurt).

- Formatting options:
  - FORMAT= applies the standard SAS formats to the column.
  - WIDTH= sets the width of the column.
  - FLOW wraps the text within the width you specified.
  - NOPRINT suppresses printing that column.
  - You can replace the label as the column heading by specifying the new heading in quotes (use a / to insert a line break).

BREAK and REBREAK statements:

- BREAK adds summaries (subtotals in this case) every time the group column(s) change.
- REBREAK gives you grand totals.
- You can specify whether the break occurs before or after the group.
- There are many options for these:
  - OL -> Overline.
  - DOL -> Double overline.
  - UL -> Underline.
  - DUL -> Double underline.
  - SUMMARIZE -> Summarize each group.
  - SKIP -> Skip a line after the break.
  - SUPPRESS -> Don?t repeat the break variable on the summary line.

```
/* general syntax for proc report */
/* create report */
proc report data=work.temp <options>;
  title <title>;
  column <vars list>;
  define <column1> / <define type and column attributes>;
  define <column2> / <define type and column attributes>;
  ...;
run;
```

5.3 Proc format

- Used to create custom formats and informats.

- Use an INVALUE statement for informats and a VALUE statement for formats.
- If the original value is a character, then you need to specify the $.
- Setting ranges:
  - a - b: a <= var <= b.
  - a <- b: a < var <= b.
  - a -< b: a <= var < b
  - a <-< b: a < var < b.
  - Can also use low and high.
  - Can also specify missing values (.=) and other values (other=).

```
/* example for proc format */
/* define format and informat */
proc format;
  value age low-13='Young' 14-16='Average' 17-20='Old' .='Not given';
  invalue $ sex 'M'='Male' 'F'='Female';
run;

/* read in data */
data work.student;
  format Age age.;
  infile '/folders/myfolders/Handbook/Data/Student data.txt';
  input Name $ Sex :$sex. Age Height Weight;
run;

/* view dataset */
proc print data=work.student;
run;
```

Obs	Age	Name	Sex	Height	Weight
1	Average	Alfred	Male	69.0	112.5
2	Young	Alice	Female	56.5	84.0
3	Young	Barbara	Female	65.3	98.0

5.4 Proc iml

IML (Interactive Matrix Language):

- The IML procedure is finished after the QUIT statement.

Syntax:

- Initialize vector -> *vector*={*values*}.
- Print vector -> print *vector*.

Operators:

- To perform calculations:
  - Addition +.
  - Subtraction -.
  - Multiplication *.
  - Division /.
  - Switch sign -.
  - Power **.
  - Multiplication (elementwise) #.
  - Power (elementwise) ##.
- To modify:
  - Subscripts [].
  - Horizontal concatenation ||.
  - Vertical concatenation //.
  - Transpose <accent mark (the one on the tilde key)>.

Functions:

- J(*nrow(, ncol, value)*):
  - Creates a matrix that has *nrow* rows, *ncol* columns, and all elements equal to *value*.
  - *ncol* and *value* arguments are optional;
- I(*dimension*):
  - Creates an identity matrix of a given size.
- REPEAT(*matrix, nrow, ncol*):
  - Creates a new matrix by repeating elements of *matrix*.
- SHAPE(*matrix, nrow(, ncol, padvalue*)):
  - Creates a new matrix by reshaping *matrix*.

  - Cycles back and repeats values to fill in the matrix when no pad value is given.
  - Cycles through *matrix* elements in row-major order and fills in the new matrix with *padvalue* after the first cycle through *matrix*.

Random number generation (example):

1. Allocate a vector with *vector*=j(*nrow(, ncol, initial value)*).
2. Fill vector with call randgen(*vector*, *distribution, parameter1, ...*):

6 Macro language

6.1 Macro definition and calling

- Define a macro with %MACRO and %MEND.
- Call a macro with %*macroname* (using a semicolon following a macro call is not recommended and can cause problems).

```
/* general syntax for a macro definition and call */
/* define macro */
%macro <macro name>;
  <macro text>;
%mend <macro name>;

/* call macro */
/* -> semicolon is included after call so that code underneath is highlighted
    correctly */
%<macro name> ;
```

6.2 Macro variables

- Help to make macros more dynamic and flexible.
- Refer to a macro variable with &*macrovar*.
- Global macro variables:
  - Defined outside of a macro with a %LET statement.
  - Exist for the remainder of the current SAS session.
- Local macro variables:
  - Defined within a macro and are not specifically defined as global.

  - Exist only during execution of the macro in which they are defined.

6.3 Macro parameters

Parameter lists are a list of macro variables referenced within the macro. There are two types: positional and keyword (can also use both at once for a mixed parameter list, but kind of confusing).

Positional parameters:

- Define macro parameters in a particular order.
- Parameter names are supplied when the macro is defined.
- Parameter values are supplied when the macro is called (and they must appear in the same order as their corresponding parameter names).

Keyword parameters:

- Assigned a default value after an equal sign.
- Can appear in any order and can be omitted from the call without placeholders.
  - If omitted from the call, a keyword parameter receives its default value.

```
/* general syntax for a macro parameters */
/* define macro with positional parameters */
%macro <macro name>(<parameter1>,<parameter2> ,...);
  <macro text>;
%mend <macro name>;

/* call macro with positional parameters */
%<macro name>(<value1>,<value2> ,...) ;

/* define macro with keyword parameters */
%macro <macro name>(<keyword1>=<value1>,<keyword2>=<value2> ,...);
  <macro text>;
%mend <macro name>;

/* call macro with keyword parameters */
%<macro name>(<value1>,<value2> ,...) ;
```

Parameter validation (two methods):

1. Use the IN operator (the MINOPERATOR option is required).
2. Use data-driven parameter validation.
   - Useful when the list of valid parameters is extremely long or changes frequently.

- Uses PROC SQL to create macro variables that contain values from a query result.

```
/* general syntax for parameter validation using in operator */
/* define macro */
%macro <macro name>(<parameter>) / minoperator;
  /* check if a valid parameter is given */
  %if &<parameter> in <list of values> %then
    <action>;
  %else
    %put <error message>;
%mend <macro name>;

/* general syntax / example for data driven parameter validation */
/* define macro */
%macro <macro name>(<parameter>) / minoperator;
  /* create list acceptable values from the given dataset */
  proc sql noprint;
    select distinct <var> into :list separated by ' '
      from work.temp;
  quit;

  /* check if a valid parameter is given */
  %if &<parameter> in &list %then
    <action>;
  %else
    %put <error message>;
%mend <macro name>;
```

6.4 Conditional processing and iterative statements

Conditionally process portions of a macro:

- Uses %IF, %THEN, and %ELSE statements.
- The macro language does not contain a subsetting %IF statement, so you cannot use %IF without %THEN.
- Difference between %IF-%THEN/%ELSE and IF-THEN/ELSE:
  - The macro ones conditionally generate text, while the other conditionally executes SAS statements during DATA step execution.

Iterative statement:

- Executes a section of a macro repetitively based on the value of an index variable.
- Uses %DO loops (so can use %TO, %UNTIL or %WHILE loops).

```
/* general syntax for conditional processsing and iterative statements */
%macro <macro name>;
  /* conditionally process this portion */
  %if <condition> %then
    <action>;
  %else
    <action>;

  /* iteratively perform this portion */
  %do i=1 %to <n> (%by <increment>);
    <action>;
  %end;
%mend <macro name>;
```

6.5 Macro functions

Here are some functions related to macro variables (they can be used outside of a DATA / PROC step)...

%EVAL(*expression*):

- Evaluates integer arithmetic or logical expressions (will return a 0 for false and a 1 for true).
- Discards the fractional part when it performs division on integers that would result in a fraction.
- Expression can only contain integers.

%INCLUDE:

- Brings a SAS programming statement, data lines, or both, into a current SAS program.
- Allows you to run another SAS program (in an external file) without having to have it all typed out in the current session (i.e. saves typing and helps organization).

%PUT:

- Writes macro variable values as text in the SAS log.
- Can include regular text as well (do not need quotes surrounding text).
- If you place an equal sign between the ampersand and the macro variable name of a direct macro variable reference, the macro variable's name is displayed in the log along with its value (e.g. %PUT &=x writes X=*value* to the log).

%SCAN(*var,k*):

- Returns the *k*th word from var (if k<0, counts from right to left).

%SUBSTR(*var,start,length*):

- Returns the parsed string of *var* (starting from *start* and *length* characters long).

%SYMGET(*macrovar*):

- Returns the value of a macro variable to the DATA step during DATA step execution.
- There are variations of SYMPUT for different types of *value*.

%CALL SYMPUT(*macrovar,value*):

- Assigns a value produced in a DATA step to a macro variable.
- *macrovar*:
  - Most useful when *macrovar* is a variable name (or an expression containing a variable name) because a unique macro variable can be created and assigned a value from each observation.
  - If *macrovar* is a character string, SYMPUT creates only one macro variable, and its value changes in each iteration of the program. Only the value assigned in the last iteration remains after program execution is finished.
- || (concatenation operator):
  - Concatenates character values.
  - Useful when making macro variable names.
- There are variations of SYMPUT for different types of *value*.

%SYSEVALF(*expression, conversion type*):

- Same thing as %EVAL, except it can have decimals in the expression and keeps the decimals in the result.
- Conversion type can be: BOOLEAN, CEIL, FLOOR or INTEGER.

%UPCASE:

- Returns a string in all caps.

6.6 Resolving macro variables

Rules for processing consecutive ampersands:

1. Two ampersands always resolve (simplify) to one ampersand.

2. The macro processor continues to read left to right until the end of the reference is reached.
    - i.e. The single ampersand that results from two from Rule 1 is left alone as the macro processor reads left to right, and will not be processed until a subsequent read.
    - So with n ampersands, ampersands one and two resolve to one, then three and four resolve to one, and so on.
    - The macro processor will then go back and re-process those results only after the entire reference has been processed initially;
3. When a reference contains consecutive ampersands, after resolution, at least one more read will follow (this is the only way to force more reads by the macro processor).

Notation and example:

- If &a resolves to b, then it can be written as &a => b.
- (n)a denotes that the macro variable a is preceded by n ampersands (n is referred to as the coefficient of the macro variable).
- For illustration purposes, assume:
    - &a => b.
    - &b => c.
    - &c => d.
    - &d => e.
    - etc.

Macro variable references of the form (n)a:

- With $n = 2$, &&a => &a => b.
    - This is because of Rule 1.
    - The only difference between &&a and &a is one more read, but the end result is the same.
    - So in this case, the extra ampersand adds no value (i.e. $n = 2$ is not a useful coefficient of a).
- With $n = 3$, &&&a => &b => c.
    - Rule 1 says that the first two resolve to a single ampersand, and Rule 2 says that the third is used to resolve a.

  - i.e. After the first read you are left with the single ampersand that came from the first two of the original reference, plus the resolution of a (which is b). A second read would then resolve b.

- So, when the possible values of a macro variable are also macro variables, repeated ampersands creates a chain effect of macro variable resolution.
  - However only some numbers of n are useful.

Patterns when the initial reference (prior to any read) shows only one ampersand:

- When n is even, the macro variable does not get resolved on that read (i.e. not useful).
- So useful coefficients are odd numbers of n that resolve to something unique.
  - (1)a => b.
  - (3)a => (1)b => c.
  - (7)a => (3)b => (1)c => d.
  - And so on...

Macro variable references of other forms:

- (n)a&b, where n is even and a is not a macro variable.
  - The effect of the even coefficient is a delayed resolution.
  - On the first read, the macro processor resolves the n ampersands to $n/2$ ampersands (because of Rule 1) and also resolves &b.
  - Thus, you are left with ($n/2$) ampersands and the name of a new macro variable, the concatenation of a and simplified &b.
  - So, useful coefficients of the form (n)a&b are equivalent to twice the useful coefficients of the form (n)a.
- There are many other helpful forms...

Another special character:

- Wherever an ampersand shows up, the macro processor will try to resolve anything that follows, including within a string of text not meant to be resolved.
- So if a macro variable reference is at the beginning of a text string...
  - Use a period as a delimiter to signify the end of a macro variable reference.

  - If multiple reads are necessary for resolving the macro variable immediately before the text string, then you need that many periods.
  - If want a period in the end result (such as a filename), then just use an extra one.

6.7 Options

Use these in OPTIONS statement to help debug macros.

- Specified options remain in effect for the rest of the SAS session or until you issue another OPTIONS statement to change the options again.

MCOMPILENOTE=ALL:

- Used to verify macro compilation.

MPRINT:

- Used to view the text generated by macro execution (gets output to SAS log).
- i.e. Translates process from a macro program to regular SAS statements.

MLOGIC:

- If a macro is invoked, this option displays messages that identify the following:
  - The beginning of macro execution.
  - Values of macro parameters at invocation.
  - Execution of each macro program statement.
  - Whether each %IF condition is true or false.
  - The ending of macro execution.
- Note -> This produces a lot of output, so only try it on test runs.

SYMBOLGEN:

- Displays the results of resolving macro variable references.
- Useful for debugging.

6.8 Macro storage

Submitting a macro definition causes the following actions:

1. The macro is compiled.
    - Macro language statements or expressions are checked for syntax errors and then compiled.
    - SAS statements and other text are not checked for syntax errors and are not compiled.
2. The macro is stored.
    - By default, the compiled macro is stored in the temporary catalog work.sasmacr, with an entry type of MACRO.

Saving macros using the stored compiled macro facility:

- Contains permanently compiled macros (and may or may not have source code stored in the same directory).
- The compiled macros can be seen in the SAS macro catalog SASMACR.
- Advantages of storing already compiled macros:
    - Decreased processing time, the macro will be the same every time it is run, and users are not able to modify the macro.
    - i.e. Reports and analysis produced with these macros will be consistent.
- Strategy for saving macro:
    - Create a folder where SAS will store the catalog SASMACR.
    - Assign a library to that folder with a LIBNAME statement.
    - Then submit the macro definitions that you want to go in there by adding the STORE option to the macros.
    - Can make the source code of the macro available by adding the SOURCE option.
    - Can also add descriptions to the macros with the DES= option.
- Strategy for using stored macros:
    - Just have to run the OPTIONS and LIBNAME statement used to save the macro.
    - Then you are able to use those macros as if they were submitted in the current session.
    - Can view all stored macros with PROC CATALOG and CONTENTS statement.
    - Can delete a macro from the catalog by using a DELETE statement.

```
/* general syntax for saving macros */
/* specify options and assign library */
options mstored sasmstore=<libname>;
libname <libname> <folder path>;

/* define and store macro */
%macro <macro name> / store source des=<description>;
  <macro text>;
%mend <macro name>;

/* view stored macros */
proc catalog cat=<libname>.sasmacr;
  contents;
run;

/* view source code of stored macro */
%copy <macro name> / source;

/* delete stored macro from catalog */
proc catalog cat=<libname>.sasmacr;
  delete <macro name> / et=macro;
run;
```

7 Reusable macros

The code below shows where these macros are stored in addition to more related code:

```
/* specify options and assign library */
options mstored sasmstore=storemac;
libname storemac '/folders/myfolders/Handbook/Macros';

/* view stored macros */
proc catalog cat=storemac.sasmacr;
  contents;
run;

/* view source code of stored macro */
%copy <macro name> / source;

/* delete stored macro from catalog */
proc catalog cat=storemac.sasmacr;
  delete <macro name> / et=macro;
run;
```

7.1 Reshape dataset from wide to long

%wideToLong:

- Purpose:
  - Reshape dataset from wide to long.
- Arguments:
  - wideData -> Input dataset (wide).
  - longData -> Output dataset (long).
  - stackVars -> List of variables to be stacked.
  - keepRow:
    - 0 (no) or 1 (yes, default).
    - Specifies whether or the row index should be kept in the output dataset.
    - Tip -> Set to 0 if transposing a row vector.
  - keepVars:
    - List of additional variables to keep in output dataset.
    - Tip -> Do not need to supply an argument if only the stacked variables are wanted.
- Call:
  - Can rename the row index, column index and newly created variable with a RENAME= option for the output dataset.
    - rowIndex -> Identifier for which observation is being read from the input dataset.
    - colIndex -> Identifier for which column (of the stacked vars) is being read from the input dataset.
    - newVar -> Newly created column of the values of the stacked vars.
    - Tip -> If keepRow=0 then do not need to RENAME= option for rowIndex.

```
/* macro to reshape dataset from wide to long */
/* definition */
%macro wideToLong(wideData=, longData=, stackVars=, keepRow=1, keepVars=);
  data &longData;
    /* specify dataset to reshape */
    set &wideData;
```

```
7
8     /* specify array for variables to be stacked */
9     array temp{*} &stackVars;
10
11    /* set index for which observation is being read from long dataset */
12    rowIndex+1;
13
14    /* transpose the desired columns and output record */
15    do colIndex=1 to dim(temp);
16      newVar=temp{colIndex};
17      output;
18    end;
19
20    /* conditioanlly modify columns to be kept */
21    %if &keepRow=1 %then
22      %do;
23        %let keepVars=&keepVars rowIndex;
24      %end;
25
26    /* attach rest of desired columns */
27    keep &keepVars colIndex newVar;
28  run;
29 %mend wideToLong;
30
31 /* call */
32 %wideToLong(wideData=work.temp,longData=work.temp2(rename=(rowIndex=<i>
       colIndex=<j> newVar=<newvar>)),
33   stackVars=<vars list>, keepRow=<0or1>, keepVars=<vars list>);
```

7.2 Get info about dataset

- Purpose:
  - Get info about dataset.
- Arguments:
  - what:
    - nobs or vars.
    - Specifies whether to perform process to view number of observations (nobs) or variables (vars).
  - inData -> Input dataset.

```
1 /* macro to get info about a dataset */
2 %macro info (what=, inData=) / minoperator;
3   /* check if a valid parameter is given */
```

```
  %if &what in (nobs vars) %then
    %do;
      /* determine which process to perform */
      %if &what=nobs %then
        %do;
          /* get table info */
          proc contents data=&inData noprint out=work.temp;
          run;

          /* extract desired info */
          proc sort data=work.temp nodupkey out=work.temp2 (keep=nobs);
            by nobs;
          run;

          /* view number of observations */
          proc print data=work.temp2 noobs;
          run;
        %end;
      %else
        %do;
          /* get table of info */
          proc contents data=&inData varnum noprint out=work.temp;
          run;

          /* extract desired info */
          proc sort data=work.temp out=work.temp2(keep=Name Varnum);
            by varnum;
          run;

          /* view variable names */
          proc print data=work.temp2 noobs;
          run;
        %end;
    %end;
    %else
      %put Error -> what must be nobs or vars;
%mend info;

/* call */
%info(what=<what>, inData=work.temp)
```

8 Simulation

8.1 Inner workings and implicit / explicit output statements

Inner workings of SAS:

- There is an implicit looping mechanism in SAS, so when no input dataset is specified, the DATA step only executes once. But when there is an input dataset with n observations, the DATA step executes n times (once through for each observation).

Implicit / explicit output statements:

- If there is no OUTPUT statement, SAS does an implicit OUTPUT statement at the end of a DATA step.
  - So if use DO loop to generate random numbers and do not have an OUTPUT statement, there will only be one observation because the DATA step is only executed once.
- So, with a DO loop you have to use an explicit OUTPUT statement, which causes SAS to write the current observation.
  - When there is an explicit OUTPUT statement, SAS no longer executes the implicit one.

8.2 Random number generation

CALL STREAMINIT(*seed*):

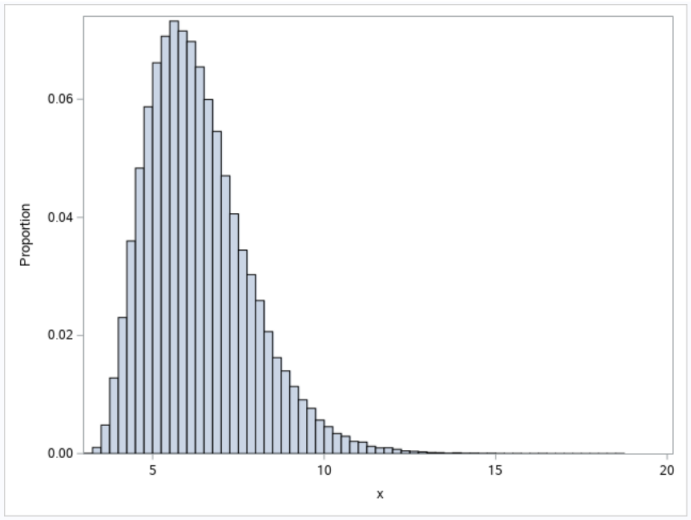
- Sets random seed.

RAND(*distribution, parameter1, ...*):

- Use this function to generate random numbers from a known distribution.
- May need to perform additional operations to add a location or rate/scale parameter.
  - Location -> RAND(<...>) + *location param.*
  - Rate -> RAND(<...>) / *rate param.*
  - Scale -> RAND(<...>) * *scale param.*
  - Note -> Scale = 1/Rate.

```
/* example for random number simulation */
/* initialize everything */
%let n=100000; /* number of random data points to generate */
%let shape=5; /* shape parameter value */
%let rate=1.5; /* rate parameter value */
%let location=3; /* location parameter value */

/* simulate data */
data work.sim;

```

```
  /* set random seed */
  call streaminit(1);

  /* generate numbers from gamma distribution with rate parameter */
  /* -> equivalent to rand('gamma',&shape)*(&scale), where scale=1/rate */
  do iteration=1 to &n;
    x=rand('gamma',&shape)/(&rate)+&location;
    output;
  end;
run;

/* plot data */
proc sgplot data=work.sim;
  histogram x / scale=proportion;
run;

/* summarize data */
proc means data=work.sim maxdec=3;
  var x;
run;
```



The MEANS Procedure

Analysis Variable : x				
N	Mean	Std Dev	Minimum	Maximum
100000	6.332	1.488	3.150	18.671

8.3 Simulating methods

General logic:

1. Figure out the model and generate many samples.
    - Model depends on the experiment you are simulating (e.g. could be a binomial experiment or values assumed to come from exponential distribution, or etc.).
    - Output after each simulation (i.e. after each sample is generated).
2. Calculate statistics for each sample.
    - Use arrays and of to perform calculations over the entire row.
3. Accumulate results over all the simulated datasets.
    - Can use PROC MEANS or PROC UNIVARIATE to summarize results. Can also plot results with PROC SGPLOT.

Data structure:

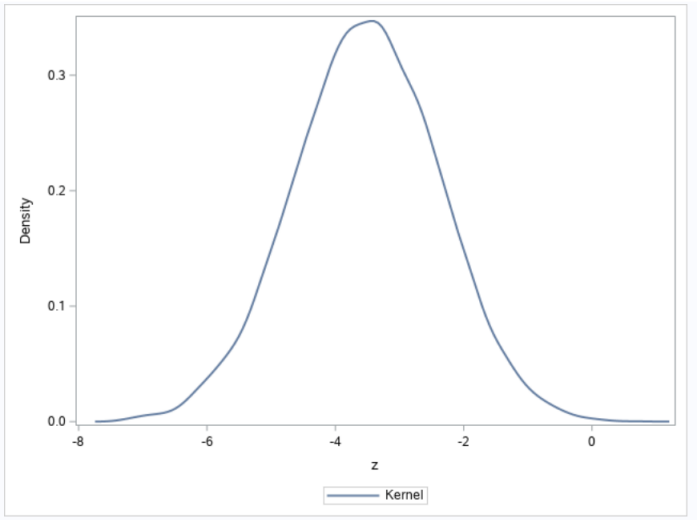
- Rows:
    - Represent the different simulations (i.e. the first row is the first sample, the second row is the second sample, ...).
    - Number of rows is equal to the number of simulations or samples.
- Columns:
    - Represent individual observations (i.e the fifth column of the third row is the fifth observation of the third sample).
    - Number of columns is equal to the sample size.

Example -> Simulation study of one-proportion z-test:

- Test whether a proportion is equal to a hypothesized proportion.
- Hypotheses -> $H_0 \rightarrow p = 0.8 \quad \& \quad H_A \rightarrow p \neq 0.8$.
- This example calculates the power of the test if the true proportion is 0.1 less than the hypothesized proportion.
- < test statistic distribution, p-value calculation formula and decision rule >.

```
/* example for simulating methods */
/* -> one proportion z-test */
/* initialize everything */
%let exp=10000; /* number of simulations to run */
%let n=200; /* sample size */
%let p=0.7; /* true proportion of success */
%let p0=0.8; /* assumed proportion of success */
%let alpha=0.10; /* significance level */

```

```
/* simulate data */
data work.sim (drop=i);

  /* set random seed */
  call streaminit(2);

  /* create observation holders */
  array x(&n);

  /* generate samples */
  do Experiment=1 to &exp;
    /* generate observations */
    do i=1 to &n;
      x(i)=rand('binomial',&p,1);
    end;
    /* output sample */
    output;
  end;
run;

/* calculate statistics for each sample */
data work.stat (drop=x1-x&n);

  /* specify simulated dataset */
  set work.sim;

  /* create observation holders */
  array x(&n);

  /* calculate  p hat, standard error, test statistic, and p value */
  pHat=sum(of x{*})/&n;
  se=sqrt(&p0*(1-&p0)/&n);
  ts=(pHat-&p0)/se;
  pValue=2*(1-probnorm(abs(ts)));

  /* make decision */
  decision=(pValue lt &alpha);
run;

/* accumulate results over all the simulated datasets */
/* plot of test statstics */
proc sgplot data=work.stat;
  density ts / type=kernel;
  xaxis label='z';
run;

/* calculate power */
proc means data=work.stat n mean;
  var decision;
run;
```



The MEANS Procedure

Analysis Variable : decision	
N	Mean
10000	0.9503000

8.4 Bootstrapping

General logic:

1. Create dataset of original sample.
    - Assign number of observations to a macro variable with SYMPUTX (use variation to remove extra blanks).
    - Transpose sample into a row vector (PREFIX option specifies prefix for created variables and VAR statement specifies the column to transpose).
2. Create bootstrap samples by sampling with replacement from the original sample (also calculate desired statistic for each sample).
3. Create confidence intervals.
    - PCTLPTS option specifies additional percentiles that aren't automatically computed by PROC UNIVARIATE.
    - PCTLPRE option specifies prefixes of variables in the output dataset.
    - Note -> These options must be used together.

```
/* example for bootstrapping */
/* -> bootstrap confidence intervals for median height */
```

```
/* read in data */
data work.student;
  infile '/folders/myfolders/Handbook/Data/Student data.txt';
  input Name $ Sex $ Age Height Weight;
run;

/* extract number of observations */
data _null_;
  set work.student nobs=temp;
  call symputx('n',temp);
run;

/* transpose dataset */
proc transpose data=work.student out=work.original(keep=Height1-Height&n)
    prefix=Height;
  var Height;
run;

/* initialize number of bootstrap samples */
%let b=10000;

/* create bootstrap samples */
data work.boot (drop=Height1-Height&n j k);

  /* specify original sample */
  set work.original;

  /* set seed */
  call streaminit(3);

  /* create original observation holders */
  array origH{*} Height1-Height&n;

  /* create bootstrap observation holders */
  array bootH{&n};

  /* generate bootstrap samples */
  do i=1 to &b;

    /* generate an individual bootstrap sample */
    do j=1 to &n;
      /* randomly select observation from original sample */
      k=rand('integer',1,&n);
      bootH{j}=origH{k};
    end;

    /* calculate statistic */
      medianH=median(of bootH{*});

    /* output sample and statistic */
```

```
    output;
  end;
run;

/* calcualte bootstrap confidence intervals */
proc univariate data=work.boot noprint;
  var medianH;
  output out=work.cis pctlpts=2.5,5,50,75,95,97.5 pctlpre=P;
run;

/* view CI limits */
title 'Confidence interval limits';
proc print data=work.cis noobs;
run;
```

Confidence interval limits

P2_5	P5	P50	P75	P95	P97_5
59	59.8	62.8	63.5	64.8	65.3

9 Miscellaneous

9.1 Libraries

```
/* general syntax for assigning a library */
libname <libname> <folder path>;
run;

/* general syntax for deleting a dataset from a library */
proc datasets library=<libname>;
  delete <dataset>;
run;

/* general syntax for deleting all datasets from a library */
proc datasets library=<libname> kill;
run;
```

9.2 Converting data types

To convert character values to numeric values, use the INPUT function.

- Syntax -> *newvar* = input(*original variable, informat.*);

To convert numeric values to character, use the PUT function:

- *newvar* = put(*original variable, format.*);

9.3 Dates

- SAS stores dates as the number of days from January 1, 1960.
- In order to tell SAS about a specific date, you need to use literals.
  - Date literals have the form 'ddmmmyyyy'd.
  - Time literals have the form 'hh:mm:ss't (based on 24 hour clock).
  - Datetime literals have the form 'ddmmmyyyy:hh:mm:ss'dt (based on 24 hour clock).

9.4 Special characters in plot titles / axis labels

- Use ODS ESCAPECHAR='~' statement.
- Then for desired string, use "~{unicode '03BB'x}" (03BB is for λ, so that will change).

9.5 Sources and resources

- New SAS documentation (Has good stuff for PROC procedures).
- ODS table names.
- Proc transpose.
- Proc report.
- Resolving macro variables.

10 Application

10.1 Example 1

```
1 /*
2 Example -> 1
3 Purpose -> Demonstrate properties of Gibbs sampling
4 */
5
6 /* open output */
7 ods rtf file='/folders/myfolders/Applications/Greaves - A second look/Output/
    Example 1.rtf'
8   bodytitle style=journal;
```

```
9
10 /* set things */
11 ods escapechar='~';
12 options mcompilenote=all;
13
14 /* macro to reshape data from wide to long */
15 /* arguments
16 -> wideData -> wide dataset
17 -> longData -> output dataset name
18 -> stackVars -> variables to be stacked
19 -> keepRow -> keep the row index in the output dataset (set to 0 if
       transposing a row vector)
20 -> keepVars -> additional variables to keep in output dataset (don't supply
       an argument if only the stacked variables are wanted) */
21 %macro wideToLong(wideData=, longData=, stackVars=, keepRow=1, keepVars=);
22   data &longData;
23     /* specify dataset to reshape */
24     set &wideData;
25
26     /* specify array for variables to be stacked */
27     array temp{*} &stackVars;
28
29     /* set index for which observation is being read */
30     rowIndex+1;
31
32     /* transpose the desired columns and output record */
33     do colIndex=1 to dim(temp);
34       newVar=temp{colIndex};
35       output;
36     end;
37
38     /* specify keep row or not */
39     %if &keepRow=1 %then
40       %do;
41         %let keepVars=&keepVars rowIndex;
42       %end;
43
44     /* attach rest of columns that we wanted to keep */
45     keep &keepVars colIndex newVar;
46   run;
47
48 %mend wideToLong;
49
50 /* macro to calculate density estimates
51 -> have to edit macro to fit situation
52 --> add/delete parameters if different than two variables of interest
53 --> edit/add conditional distribution calculations */
54 /* arguments of macro
55 -> inData -> simulated dataset
56 -> outData -> output datset names
```

```
-> vars -> variables to estimate
-> nVars -> number of variables to estimate (nVars=1,2,3,...)
-> start{i} -> first value of the ith variable to estimate {start{i} in range
     of (ith variable | other parameters) and less than stop{i}}
-> stop{i} -> last value of the ith variable to estimate {stop{i} in range of
     (ith variable | other parameters) and greater than start{i}}
-> by{i} -> increment amount for value of ith random variable value {0 < by{i
    } < stop{i}-start{i}} */
%macro estimate(inData=, outData=, vars=, nVars=, start1=, stop1=, by1=,
    start2=, stop2=, by2=);
  /* repeat process for each variable that needs estimated */
  %do v=1 %to &nVars;

    /* set variable */
    %let var=%scan(&vars, &v);

    /* intermediate calculation for density estimates */
    data work.temp&v.a;
      /* specify simulated dataset */
      set &inData;

      /* initialize value of random variable */
      value=&&start&v;

      /* specify array for new calculations */
      array temp{%sysevalf((&&stop&v-&&start&v)/&&by&v,floor)}
        k1-k%sysevalf((&&stop&v-&&start&v)/&&by&v,floor);

      /* conditionally do calculations */
      %if (&var=x) %then
        %do;

          /* loop */
          do i=1 to dim(temp);

            /* calculate f(x|lambda) for each lambda */
            temp{i}=lambdaSim*exp(-lambdaSim*value);

            /*increase RV value */
            value+&&by&v;
          end;
        %end;
      %else
        %do;

          /* loop */
          do i=1 to dim(temp);

            /* calculate f(lambda|x) for each x */
            temp{i}=(xSim+&beta)**(&alpha+1)*(value**&alpha)*exp(-value*(xSim
```

```
   +&beta))/gamma(&alpha+1);
104
105             /* increase RV value */
106             value+&&by&v;
107           end;
108         %end;
109
110       /* clean up */
111       keep iteration k1-k%sysevalf((&&stop&v-&&start&v)/&&by&v,floor);
112     run;
113
114     /* calculate final density estimates of x by averaging over lambdas */
115     proc means data=work.temp&v.a mean nonobs noprint;
116       var k1-k%sysevalf((&&stop&v-&&start&v)/&&by&v,floor);
117       output out=work.temp&v.b (drop=_type_ _freq_)
118         mean=k1-k%sysevalf((&&stop&v-&&start&v)/&&by&v,floor);
119     run;
120
121     /* reshape dataset */
122     %wideToLong(wideData=work.temp&v.b,
123       longData=work.temp&v.c(rename=(colIndex=k newVar=Estimate)),
124       stackVars=k1-k%sysevalf((&&stop&v-&&start&v)/&&by&v,floor), keepRow=0,
   keepVars=);
125
126     /* specify output dataset */
127     %let out=%scan(&outData, &v);
128
129     /* create dataset with density estimates and corresponding inputs of x */
130     data work.&out;
131       /* specify dataset with density estimates */
132       set work.temp&v.c (rename=(Estimate=f_&var.Hat));
133
134       /* calculate inputs for x */
135       if k=1 then
136         &var=&&start&v;
137       else
138         &var+&&by&v;
139
140       /* clean up */
141       drop k;
142     run;
143
144   %end;
145
146   /* delete intermediate datasets from library */
147   %do v=1 %to &nVars;
148     proc datasets library=work nolist;
149       delete temp&v.a temp&v.b temp&v.c;
150     run;
151   %end;
```

```
%mend estimate;

/* initialize everything */
%let n=100000; /* number of random data points to generate */
%let alpha=5; /* paramater value for prior */
%let beta=100; /* paramater value for prior */
%let burnin=500; /* number of burn-in iterations to use */
%let int=300; /* number of values to interpolate with for actual density
    curves */

/* simulate data */
data work.sim;

  /* set random seed */
  call streaminit(1);

  /* initialize starting values and set counter */
  xSim=5;
  lambdaSim=1.5;
  iteration=1;
  output;

  /* sample from conditional distributions */
  do iteration=2 to &n;
    xSim=rand('exponential')/lambdaSim;
    lambdaSim=rand('gamma', &alpha+1)/(xSim+&beta);
    output;
  end;

  /* remove burn-in iterations */
  if iteration>&burnin;
run;

/* calculate density estimates */
%estimate(inData=work.sim, outData=estX estLambda, vars=x lambda,
  nVars=2, start1=0, stop1=300, by1=2, start2=0, stop2=.3, by2=0.004);

/* create data for marginal density curve of x */
data work.plotX;
  /* initialize values */
  i=1;
  x2=&beta;

  /* generate pdf values */
  do until (i>&int);

    /* calculate x and f(x) */
    x=x2-&beta;
    f_x=pdf('pareto', x2, &alpha, &beta);

```

```
201     /* write observation */
202     output;
203
204     /* increase random variable value */
205     x2+1;
206
207     /* increase counter */
208     i+1;
209   end;
210   /* clean up */
211   drop x2 i;
212 run;
213
214 /* create data for marginal density curve of lambda */
215 data work.plotLambda;
216   /* initialize values */
217   i=1;
218   lambda=0;
219
220   /* generate pdf values */
221   do until (i>&int);
222
223     /* calculate f(lambda) */
224     f_lambda=pdf('gamma', lambda, &alpha, 1/&beta);
225
226     /* write observation */
227     output;
228
229     /* increase random variable value */
230     lambda+0.001;
231
232     /* increase counter */
233     i+1;
234   end;
235   /* clean up */
236   drop i;
237 run;
238
239 /* combine data for x */
240 data work.bothX;
241   set work.plotX (rename=(x=x1)) work.estX (rename=(x=x2)) ;
242 run;
243
244 /* combine data for lambda */
245 data work.bothLambda;
246   set work.plotLambda (rename=(lambda=lambda1)) work.estLambda (rename=(
      lambda=lambda2)) ;
247 run;
248
249 /* specify last i values for scatterplot */
```

```
%let i=100;

/* create scatter plot of simulated values of x */
proc sgplot data=work.sim;
  scatter x=iteration y=xSim;
  where iteration>&n-&i;
  title 'Figure 1: Scatterplot of generated sample values of X';
  xaxis label='Iteration';
  yaxis max=100 label='x';
run;

/* create scatter plots of simulated values of lambda */
proc sgplot data=work.sim;
  scatter x=iteration y=lambdaSim;
  where iteration>&n-&i;
  title "Figure 2: Scatterplot of generated sample values of ~{unicode '03BB'
    x}";
  xaxis label='Iteration';
  yaxis label="~{unicode '03BB'x}";
run;

/* create histogram of x */
proc sgplot data=work.sim;
  histogram xSim / scale=proportion nbins=250;
  title 'Figure 3: Histogram of generated sample values of X';
  xaxis max=200 label='x';
run;

/* create histogram of lambda */
proc sgplot data=work.sim;
  histogram lambdaSim / scale=proportion nbins=250;
  title "Figure 4: Histogram of generated sample values of ~{unicode '03BB'x
    }";
  xaxis max=0.3 label="~{unicode '03BB'x}";
run;

/* create plot of estimated xs with marginal density curve overlaid */
proc sgplot data=work.bothX;
  series x=x1 y=f_x / legendlabel='Actual density';
  scatter x=x2 y=f_xHat / legendlabel='Estimated density';
  title 'Figure 5: Estimated and actual density of X';
  xaxis label='x';
  yaxis label='f(x)';
run;

/* create plot of estimated lambdas with marginal density curve overlaid */
proc sgplot data=work.bothLambda;
  series x=lambda1 y=f_lambda / legendlabel='Actual density';
  scatter x=lambda2 y=f_lambdaHat / legendlabel='Estimated density';
  title "Figure 6: Estimated and actual density of ~{unicode '03BB'x}";
```

```
  xaxis label="~{unicode '03BB'x}";
  yaxis label="f(~{unicode '03BB'x})";
run;

/* close output */
ods rtf close;
```

10.2 Example 2

```
/*
Example -> 1
Purpose -> Demonstrate properties of Gibbs sampling
*/

/* open output */
ods rtf file='/folders/myfolders/Applications/Greaves - A second look/Output/
    Example 1.rtf'
  bodytitle style=journal;

/* set things */
ods escapechar='~';
options mcompilenote=all;

/* macro to reshape data from wide to long */
/* arguments
-> wideData -> wide dataset
-> longData -> output dataset name
-> stackVars -> variables to be stacked
-> keepRow -> keep the row index in the output dataset (set to 0 if
    transposing a row vector)
-> keepVars -> additional variables to keep in output dataset (don't supply
    an argument if only the stacked variables are wanted) */
%macro wideToLong(wideData=, longData=, stackVars=, keepRow=1, keepVars=);
  data &longData;
    /* specify dataset to reshape */
    set &wideData;

    /* specify array for variables to be stacked */
    array temp{*} &stackVars;

    /* set index for which observation is being read */
    rowIndex+1;

    /* transpose the desired columns and output record */
    do colIndex=1 to dim(temp);
      newVar=temp{colIndex};
      output;
    end;

    /* specify keep row or not */
```

```
    %if &keepRow=1 %then
      %do;
        %let keepVars=&keepVars rowIndex;
      %end;

    /* attach rest of columns that we wanted to keep */
    keep &keepVars colIndex newVar;
  run;

%mend wideToLong;

/* macro to calculate density estimates
-> have to edit macro to fit situation
--> add/delete parameters if different than two variables of interest
--> edit/add conditional distribution calculations */
/* arguments of macro
-> inData -> simulated dataset
-> outData -> output datset names
-> vars -> variables to estimate
-> nVars -> number of variables to estimate (nVars=1,2,3,...)
-> start{i} -> first value of the ith variable to estimate {start{i} in range
     of (ith variable | other parameters) and less than stop{i}}
-> stop{i} -> last value of the ith variable to estimate {stop{i} in range of
     (ith variable | other parameters) and greater than start{i}}
-> by{i} -> increment amount for value of ith random variable value {0 < by{i
    } < stop{i}-start{i}} */
%macro estimate(inData=, outData=, vars=, nVars=, start1=, stop1=, by1=,
    start2=, stop2=, by2=);
  /* repeat process for each variable that needs estimated */
  %do v=1 %to &nVars;

    /* set variable */
    %let var=%scan(&vars, &v);

    /* intermediate calculation for density estimates */
    data work.temp&v.a;
      /* specify simulated dataset */
      set &inData;

      /* initialize value of random variable */
      value=&&start&v;

      /* specify array for new calculations */
      array temp{%sysevalf((&&stop&v-&&start&v)/&&by&v,floor)}
        k1-k%sysevalf((&&stop&v-&&start&v)/&&by&v,floor);

      /* conditionally do calculations */
      %if (&var=x) %then
        %do;

```

```
86          /* loop */
87          do i=1 to dim(temp);
88
89            /* calculate f(x|lambda) for each lambda */
90            temp{i}=lambdaSim*exp(-lambdaSim*value);
91
92            /*increase RV value */
93            value+&&by&v;
94          end;
95        %end;
96      %else
97        %do;
98
99          /* loop */
100         do i=1 to dim(temp);
101
102           /* calculate f(lambda|x) for each x */
103           temp{i}=(xSim+&beta)**(&alpha+1)*(value**&alpha)*exp(-value*(xSim
    +&beta))/gamma(&alpha+1);
104
105           /* increase RV value */
106           value+&&by&v;
107         end;
108       %end;
109
110     /* clean up */
111     keep iteration k1-k%sysevalf((&&stop&v-&&start&v)/&&by&v,floor);
112   run;
113
114   /* calculate final density estimates of x by averaging over lambdas */
115   proc means data=work.temp&v.a mean nonobs noprint;
116     var k1-k%sysevalf((&&stop&v-&&start&v)/&&by&v,floor);
117     output out=work.temp&v.b (drop=_type_ _freq_)
118       mean=k1-k%sysevalf((&&stop&v-&&start&v)/&&by&v,floor);
119   run;
120
121   /* reshape dataset */
122   %wideToLong(wideData=work.temp&v.b,
123     longData=work.temp&v.c(rename=(colIndex=k newVar=Estimate)),
124     stackVars=k1-k%sysevalf((&&stop&v-&&start&v)/&&by&v,floor), keepRow=0,
    keepVars=);
125
126   /* specify output dataset */
127   %let out=%scan(&outData, &v);
128
129   /* create dataset with density estimates and corresponding inputs of x */
130   data work.&out;
131     /* specify dataset with density estimates */
132     set work.temp&v.c (rename=(Estimate=f_&var.Hat));
133
```

```
      /* calculate inputs for x */
      if k=1 then
        &var=&&start&v;
      else
        &var+&&by&v;

      /* clean up */
      drop k;
    run;

  %end;

  /* delete intermediate datasets from library */
  %do v=1 %to &nVars;
    proc datasets library=work nolist;
      delete temp&v.a temp&v.b temp&v.c;
    run;
  %end;
%mend estimate;

/* initialize everything */
%let n=100000; /* number of random data points to generate */
%let alpha=5; /* paramater value for prior */
%let beta=100; /* paramater value for prior */
%let burnin=500; /* number of burn-in iterations to use */
%let int=300; /* number of values to interpolate with for actual density
    curves */

/* simulate data */
data work.sim;

  /* set random seed */
  call streaminit(1);

  /* initialize starting values and set counter */
  xSim=5;
  lambdaSim=1.5;
  iteration=1;
  output;

  /* sample from conditional distributions */
  do iteration=2 to &n;
    xSim=rand('exponential')/lambdaSim;
    lambdaSim=rand('gamma', &alpha+1)/(xSim+&beta);
    output;
  end;

  /* remove burn-in iterations */
  if iteration>&burnin;
run;
```

```
183
184 /* calculate density estimates */
185 %estimate(inData=work.sim, outData=estX estLambda, vars=x lambda,
186   nVars=2, start1=0, stop1=300, by1=2, start2=0, stop2=.3, by2=0.004);
187
188 /* create data for marginal density curve of x */
189 data work.plotX;
190   /* initialize values */
191   i=1;
192   x2=&beta;
193
194   /* generate pdf values */
195   do until (i>&int);
196
197     /* calculate x and f(x) */
198     x=x2-&beta;
199     f_x=pdf('pareto', x2, &alpha, &beta);
200
201     /* write observation */
202     output;
203
204     /* increase random variable value */
205     x2+1;
206
207     /* increase counter */
208     i+1;
209   end;
210   /* clean up */
211   drop x2 i;
212 run;
213
214 /* create data for marginal density curve of lambda */
215 data work.plotLambda;
216   /* initialize values */
217   i=1;
218   lambda=0;
219
220   /* generate pdf values */
221   do until (i>&int);
222
223     /* calculate f(lambda) */
224     f_lambda=pdf('gamma', lambda, &alpha, 1/&beta);
225
226     /* write observation */
227     output;
228
229     /* increase random variable value */
230     lambda+0.001;
231
232     /* increase counter */
```

```
    i+1;
  end;
  /* clean up */
  drop i;
run;

/* combine data for x */
data work.bothX;
  set work.plotX (rename=(x=x1)) work.estX (rename=(x=x2)) ;
run;

/* combine data for lambda */
data work.bothLambda;
  set work.plotLambda (rename=(lambda=lambda1)) work.estLambda (rename=(
    lambda=lambda2)) ;
run;

/* specify last i values for scatterplot */
%let i=100;

/* create scatter plot of simulated values of x */
proc sgplot data=work.sim;
  scatter x=iteration y=xSim;
  where iteration>&n-&i;
  title 'Figure 1: Scatterplot of generated sample values of X';
  xaxis label='Iteration';
  yaxis max=100 label='x';
run;

/* create scatter plots of simulated values of lambda */
proc sgplot data=work.sim;
  scatter x=iteration y=lambdaSim;
  where iteration>&n-&i;
  title "Figure 2: Scatterplot of generated sample values of ~{unicode '03BB'
    x}";
  xaxis label='Iteration';
  yaxis label="~{unicode '03BB'x}";
run;

/* create histogram of x */
proc sgplot data=work.sim;
  histogram xSim / scale=proportion nbins=250;
  title 'Figure 3: Histogram of generated sample values of X';
  xaxis max=200 label='x';
run;

/* create histogram of lambda */
proc sgplot data=work.sim;
  histogram lambdaSim / scale=proportion nbins=250;
  title "Figure 4: Histogram of generated sample values of ~{unicode '03BB'x
```

```
    }";
  xaxis max=0.3 label="~{unicode '03BB'x}";
run;

/* create plot of estimated xs with marginal density curve overlaid */
proc sgplot data=work.bothX;
  series x=x1 y=f_x / legendlabel='Actual density';
  scatter x=x2 y=f_xHat / legendlabel='Estimated density';
  title 'Figure 5: Estimated and actual density of X';
  xaxis label='x';
  yaxis label='f(x)';
run;

/* create plot of estimated lambdas with marginal density curve overlaid */
proc sgplot data=work.bothLambda;
  series x=lambda1 y=f_lambda / legendlabel='Actual density';
  scatter x=lambda2 y=f_lambdaHat / legendlabel='Estimated density';
  title "Figure 6: Estimated and actual density of ~{unicode '03BB'x}";
  xaxis label="~{unicode '03BB'x}";
  yaxis label="f(~{unicode '03BB'x})";
run;

/* close output */
ods rtf close;
```

10.3 Final report

A second look at Implementation of Gibbs Sampling within Bayesian Inference and its Applications in Actuarial Science

Colton Gearhart

Abstract

This report discusses one use of a Bayesian approach within an actuarial context. Specifically, we will investigate a particular Markov chain Monte Carlo method, Gibbs sampling. This technique enables complicated actuarial models to be analyzed by reducing them to simpler models. The properties of Gibbs sampling are demonstrated and then applied within SAS. The implementation of this algorithm involved several macros, which utilized a variety of programming concepts. Ultimately, we were able to estimate a density that does not have a closed form solution.

Introduction

Simulation is an important tool for actuaries when they are attempting to study stochastic models. Often, actuarial models are quite complicated, if not impossible to find closed form solutions for. If this is the case, simulation becomes a valuable resource. When we combine Monte Carlo simulation techniques with Bayesian analysis, it enables us to learn about the entire distribution of a quantity. In addition, this approach allows for modelling uncertainty by assigning probability distributions to parameters, not just the data. This is worked into our models via the prior distributions of our parameters, which are then updated with data. In effect, as the simulation continues, we are supplying more and more information to our parameters. This ultimately increases reliability of our estimates. Finally, in order to estimate the target integrals that we could not sample from directly, we can apply discrete formulas to our sample.

One popular method of performing this process is Gibbs sampling. Rather than sampling from the complicated multivariate posterior distributions (which may not even be possible), Gibbs sampling generates random draws from the full conditional distributions. After each draw is made, the conditional distributions of the variables get updated with the new information. This is shown by the basic Gibbs sampling algorithm shown below (*j* represents the iteration count):

0) $x^{(j)} = \left(x_1^{(j)}, x_2^{(j)}, \dots, x_k^{(j)}\right)$, with $j = 0$.
1) Set $j = j + 1$.
2) Simulate $x_1^{(j)} \sim f\left(x_1 | \, x_2^{(j-1)}, x_3^{(j-1)}, \dots, x_k^{(j-1)}\right)$.
3) Simulate $x_2^{(j)} \sim f\left(x_2 | \, x_1^{(j-1)}, x_3^{(j-1)}, \dots, x_k^{(j-1)}\right)$.

… …

k) Simulate $x_k^{(j)} \sim f\left(x_k | x_1^{(j-1)}, x_2^{(j-1)}, \dots, x_{k-1}^{(j-1)}\right)$.

k+1) Form $x^{(j)} = \left(x_1^{(j)}, x_2^{(j)}, \dots, x_k^{(j)}\right)$.

k+2) Return to step 1.

In essence, each draw depends on the previous one. In addition, with other parameters being set at their current values (i.e. they are treated as fixed), these joint densities often simplify to known distributions. This result makes many problems much easier.

The goal of this project is to investigate the properties of Gibbs sampling, a Markov chain Monte Carlo method, in addition to applying it on an example. Specifically, we will look at how a basic Gibbs sampling algorithm can be implemented within SAS.

Methods

Simulate

In order to demonstrate the properties of Gibbs sampling, we will look at an example involving the size of claims. We start by assuming the size of a claim *X* is exponentially distributed given rate parameter λ. Additionally, let λ be a random variable following a gamma distribution with parameters α and β, both of which are constants. After deriving the conditional distribution for $\lambda|x$, we find that it follows a gamma distribution with parameters α+1 and x+β. We can now implement a Gibbs sampling algorithm in SAS.

In order to do this, we first set the number of random numbers to generate (*n*), the number of burn-in iterations to be used (*burnin*), and values of parameters as macro variables (*alpha, beta*). Burn in iterations are used to allow the Markov chains to converge. Next, we initialized starting values of *x*, λ and the counter variable within a data step. Then a do loop generated *n* random numbers for *x* and λ from the dependent sampling scheme defined in the introduction. Lastly, we removed the burn-in iterations using an if statement. With the data simulated, scatter plots of the last 100 generated values for *x* and λ were created using *proc sgplot*. Histograms of the simulated data were also created using *proc sgplot*.

Estimate

We can now approximate the desired integrals by applying discrete formulas to our samples. For our marginal distribution of interest $f(x) = \int f(x|\lambda) f(\lambda) d\lambda$, we can obtain an estimate of the actual value *f(x)* at point *x* using $f(\hat{x}) = \frac{1}{n-burnin} \sum_{i=burnin}^{n} f(x|\lambda_i)$. If we do this process over many values in the range of *X*, we can obtain an estimated density of *f(x)*. Similar calculations can be done for that of λ. To do this in SAS, macros were utilized. The *estimate* macro takes the following arguments as parameters: the simulated dataset, the name of the output dataset, variables to estimate, the number of variables to estimate, and information about which values to estimate each variable at. For this example, here is how the macro

works. Within a data step, it initializes the starting value of the random variable *X* and creates an array with a different variable for each value *x* that we are going to estimate at. Next, it uses a do loop to create a dataset where each observation is calculated according to $f(x|\lambda_i)$. It then utilizes *proc means* to create a wide dataset of the final density estimates of *X* at point *x* by averaging over all of the λs.

In order to get the data to a more manageable form, another macro is used. The *wideToLong* macro takes the following arguments as parameters: the wide dataset, the long dataset to be created, the variables to stack, an indicator to keep the row index or not, and which additional variables to keep from the wide dataset. Upon being called, this macro returns the desired reshaped dataset. Finally, the last step in the *estimate* macro is to combine final density estimates of *X* with the corresponding inputs.

Iterative do statements, conditionally processing, and the technique of resolving macro variables allow this entire process to be repeated for each of the variables specified when calling the *estimate* macro. So, for this example, calling this macro resulted in two datasets of estimated densities, one for *X* and one for λ. This macro was designed to be very generalized and can be used again for the later example. There are some minor cavoites in terms of the generalization though. The conditional distributions used in the intermediate calculations need to be manually edited to fit the given situation; additionally, parameters that correspond to information about which values to estimate at may need to be added or taken away based on the number of random variables in the given situation. This macro is also efficient in terms of created datasets as there is only one dataset output for each random variable (all intermediate datasets are removed from the work library using *proc datasets*).

Validate

In practice, the marginal distributions of interest often cannot be solved for in closed form, which is why Gibbs sampling is of use. However, for this example, we chose distributions such that there would be closed-form solution so that our results can be compared. We know that $X \sim Pareto(\alpha, \beta)$ and $\lambda \sim gamma(\alpha, \beta)$. In order to plot these marginal densities with *proc sgplot*, we had to create datasets of the pdf values at many input values. Then the *series* statement in *proc sgplot* can interpolate along these points to form the marginal density curves. These were overlaid on the same plots of the estimated densities of *X* and λ, respectively.

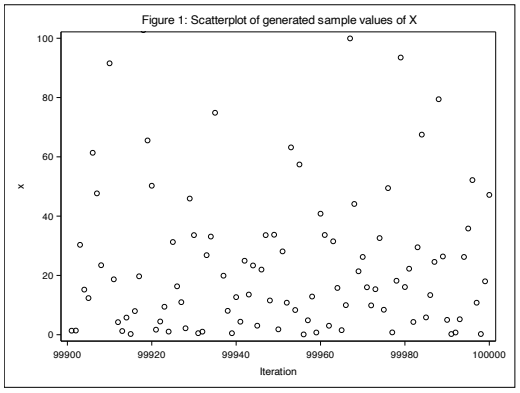
Apply

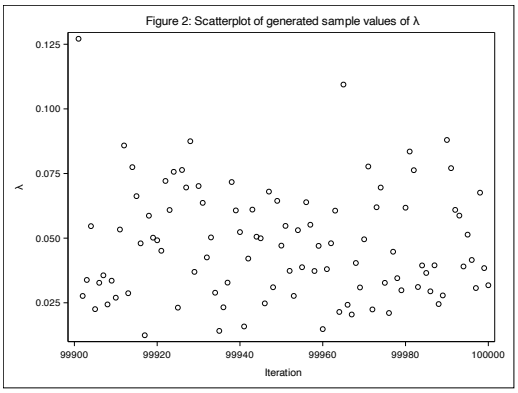
Utilizing the *estimate* macro, we could very easily perform another Gibbs sampling algorithm. Consider the discrete situation where we would like to model number of claims. Let the probability of filing a claim *P* follow a beta distribution with parameters α and β. Additionally, let the number of policies in a given portfolio *N* follow a zero-truncated Poisson distribution with parameter λ. Finally, let the number of policies that file a claim follow a binomial distribution with parameters *N* and *P*. In this scenario, *f(x)* cannot be solved for in closed form. So, in order to estimate it, we will utilize Gibbs sampling.

Using the same code from the prior example, only a few changes needed to be made to fit this scenario. The simulated random numbers needed to come from conditional binomial, beta and Poisson distributions, respectively. Then, because we are interested in the marginal density of *X*, the typical number of policies that generated a claim within an arbitrary portfolio, we needed to change the conditional distribution that SAS will average over. After doing this and initializing the macro with the correct parameters, we called the *estimate* macro. This resulted in a density estimate for the marginal distribution of *X*.

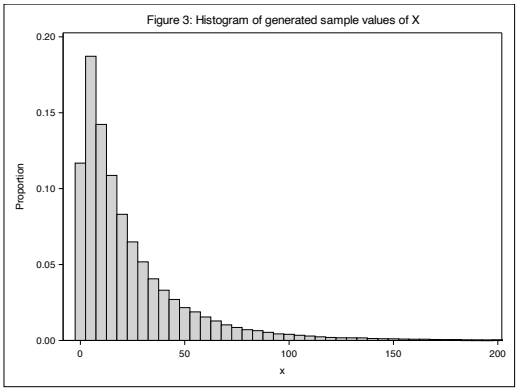
Results

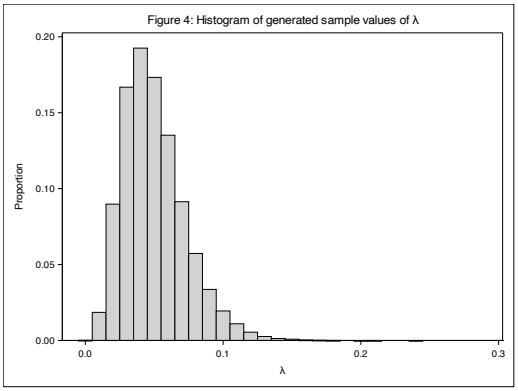
After simulating the random numbers for the first example, we wanted to check and make sure that they appeared to be a random sample. Scatter plots of the last 100 generated values of *X* and λ, respectively, were created to check this. These are shown below in Figures 1 and 2.



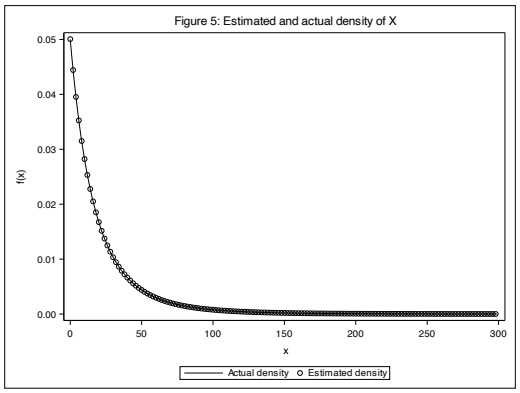


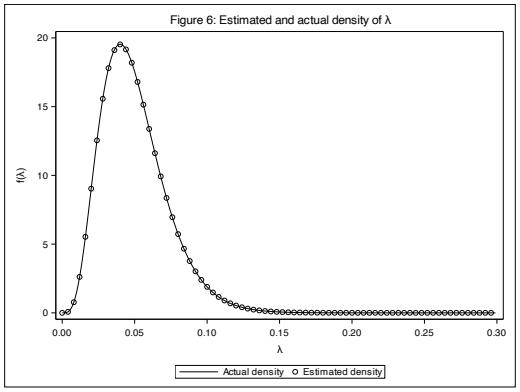
These plots indicate that there is no pattern among the generated random numbers. Thus, we can consider them to be independent random samples. Next, we created histograms of the generated samples to look at the overall distributions of each. These are shown in Figures 3 and 4.





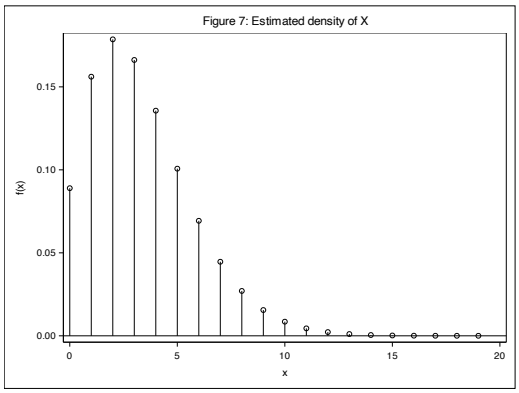
Finally, plots of the estimated densities of X and λ with their respective actual marginal densities overlaid were created. These are shown in Figures 5 and 6.





It appears that the respective marginal densities line up very well with their estimated densities. If looking at more complicated models, these results suggest that we can use Gibbs sampling to study random samples from unknown distributions.

This process was then applied to a different situation in which we wanted to model the number of policies that file a claim within an arbitrary portfolio. The estimated density of this is shown in Figure 7.



With these results, actuaries can compute several statistics for a portfolio with parameters p and n, which could ultimately help improve their predictions for the expected number of future claims.

Conclusion

Overall, this project had the goal of demonstrating the properties of Gibbs sampling and providing a possible application. Specifically, we wanted to show how this could be implemented within SAS. Future work could include investigating how many burn-in iterations

are necessary for the Markov chains to converge or why the conditional distributions only need to be known up to a normalizing constant. One remaining question that we have relates to the idea of convergence; we are not sure exactly how this works. For example, how are the generated numbers “less random” after many iterations? We suspect this knowledge to come to us in the upcoming semesters.

References

Gearhart, C., Implementation of Gibbs Sampling within Bayesian Inference and its Applications in Actuarial Science, SIAM Undergraduate Research Online, Vol. 11.

SAS Institute Inc. 2015. SAS/IML® 14.1 User’s Guide. Cary, NC: SAS Institute Inc.