Paper PP01

Efficiently Simulating Normally Distributed Data For Animation

Ben White, GlaxoSmithKline, London, UK

ABSTRACT

An exploration of efficiency strategies through the simulation of a normally distributed dataset for animation. Creating an animation of this type in SAS requires a dataset of size in excess of 500,000 observations. I will briefly touch upon the features and downsides of my initial approach then move on to discussing the process of implementing efficiency strategies. Including: How to calculate the runtime of different procedures/data manipulations allowing you to focus efforts. Utilising proc IML to initialise a matrix of random values and do the frequency count prior to the SAS loop reducing the number of steps inside the loop. I will also explore leveraging tools beyond SAS by completing a runtime comparison between R and SAS to see which programming language can produce the dataset/animation most efficiently but also to explore tools for creating animations beyond SAS

INTRODUCTION

This paper is intended to help other beginners approaching simulation for the first time not to fall into inefficient ways of simulating data. Creating animations was also a completely new process for me and was the driving force behind my investigation of efficient simulation due to the unique requirements of simulated data for animation. The main requirement being the dataset must retain the values of all previous iterations for the purposes of the animation. This is the key requirement for animations and why simulating them efficiently is so important, as the resulting dataset will become of large size with relatively few iterations. In addition the simulation will compute a Y-axis Frequency variable, I will discuss this variable in more detail in the paper, but it's purpose is to avoid the columns of dots associated with dot plots. The final data will be used to create a time series of plots which will then be animated. I wanted to use this opportunity to explore languages beyond SAS. As a predominately SAS programmer I often hear about the efficiencies working with R brings and I wanted to put that premise to the test. Note our efficient SAS simulation code does require the SAS/IML module.

THE AXIOMS OF EFFICIENT SIMULATION

1. Avoid Macro Do Loops— If you are simulating using macro do loops stop! Macro loops call the procedure or data step for the number of iterations required so suffer from high overhead for minimal work each procedure or datastep only computes a small number of values looped over a large number of iterations.

2. Vectorise Your Program — Anyone who uses R or has spent any time with a programmer who uses R…. will be aware that matrix and vector operations are more efficient than scalar operations. How do we identify scalar operations which could be vectorised? We need to look for a sum operation which could be turned into a vector product turning a loop of scalar operations into one vector operation can drastically increase the efficiency of your simulation.

3. Profile to Identify Trouble Spots — Don't guess which parts of your program are causing your simulation to run slowly. In SAS you can use the TIME function to profile a program and measure the execution time of different sections of your program. This way you can make sure the subsection is worth taking the time to make more efficient. Other programming languages have ways of profiling the program as well.

4. Pre-allocating Arrays When Looping— One of the main ideas behind efficient simulation is to reduce the number of operations, therefore generating a matrix of results one row at a time using concatenation operators should be avoided. Instead allocate space for results prior to the loop that computes the results. Similarly, if you need to simulate an array of random numbers do this in a single call.

5. Reduce the Number of Simulations— How many simulations do you really need? Is 10,000 really required perhaps 1,000 is sufficient. Always use a small number of simulations while you develop and profile a program. After you are confident your program is working as intended then increase the simulation for a final run.

6. Estimating the Run Time — Estimate the run time before you run the program this tip won't make your simulation run faster but it is important to bear in mind. Simulation tend to scale linearly with the number of iterations so if you perform a test run with 10 iterations and time how long it takes if you run 1,000 iteration the final simulation should take 100 times longer than the test run.

WORKING TOWARDS AN EFFICIENT SIMULATION IN SAS

Approaching simulating data for the first time in SAS can be quite daunting and trying to do it as efficiently as possible requires the programmer to implement some new strategies. Firstly, we need to understand what causes simulations to run slowly. The basic premise is to have a smaller number of data steps/procedures doing more work. To illustrate how to program an efficient simulation let us first show an example of how not to simulate data:

Originally, I started my simulation from just one row of data while this may be intuitive for a dynamic simulation, but we are already suffering from a high overhead to load as we have used one datastep to create one line of data.

```
data &dset.1;
 input ID $ COUNT;
 datalines;
A1 1
run;
```

I have not included the code inside the loop, but I discuss the features of the loop and their drawbacks. The full code can be found in Appendix 1. By their very nature Macro loops suffer from high overhead to work ratio, as for every iteration 5 data steps/procedures are called within the loop. Each additional data step/procedure takes time, even if they only take 0.05 seconds over 1000 iterations that is 50 seconds. Currently the program computes one normally distributed value per iteration and then compute a resource intensive row by row compare to calculate the Y-axis frequency variable. Each iteration creates a dataset of increasing size taking longer to read in and append onto.

```
%macro Jloop(n=,dset=);

%do i=2 %to &n;
...
...
...
...
...

%end;

%mend;
%Jloop(n=&n,dset=&dset);
```

To make our simulation more efficient we will focus on decreasing the number of data steps and procedures and avoiding macro do loops. The previous code called in region of 5000 data steps/procedures per 1000 iterations the efficient code below calls 5 data steps/procedures in total regardless of the number of iterations.

```
%let n=1000;                          * n is the number of data points;
```

SAS/IML stands for interactive matrix language and to give SAS users the ability to form datasets using vector operations. Vector operations have an efficiency advantage over scalar operations as we come back to the overhead to work ratio completing one operation on two columns of data is more efficient that a row by row scalar operation. By utilising proc IML we can generate all the normally distributed values we need for the animation with one call.

```
proc iml;
 names={index x Xround};              * Naming of variables in the sim1 dataset;
 i = T(1:&n);                         * Creation of a i index matrix;
 X1 = j(&n,1);                        * Initialise 1 by n marix;
 call randseed(1234);                 * Ensure the same values are produced each run;
 call randgen(X1,'normal',10,3);      * Generate n norm values with mean=10 and sd=3 ;
 XR=round(X1,1);                      * Round normally distributed data;
 X2=insert(i,X1,0,2);                 * Combine i and X1 matrices;
 X =insert(X2,XR,0,3);                * Insert rounded values also ;
 create sim1 from X [colname=names]; * Create sim1 dataset;
  append from X;
 close sim1;
quit;

proc sort data=sim1 out=sim2;         * Sort prior to Y calculation;
 by Xround index;
run;
```

The Xround variable groups the values and using the by/retain avoids doing a cell by cell comparison to create the frequency variable Y. Note this is a scalar operation, but it is only done on a relatively small dataset of size n.

```
data sim3;                            * Datastep to calculate the Y value;
 set sim2;
 by Xround;                           * Using a by statement avoids a loop;
 retain y;
 if first.Xround then Y=1;            * Assign the number of values which fall into
the;
 else Y=Y+1;                          * Xround category for ascending index;
run;
```

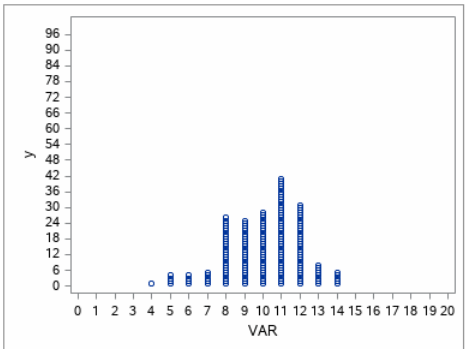
There is only one do loop in the simulation, outputting each frame into a single dataset using the 'n' variable as the ID for each frame this is the first time, we are creating a dataset of size greater than the total number of iterations. Using a BY value of 5 also reduces the size of the resulting dataset even though the impact on the efficiency of the simulation is negligible it will speed up the creation of the animation.

```
data sim4;                            * Create the final dataset for the animation;
 set sim3;
 do k=5 to &n by 5;                   * Each loop adds an additional 5 values;
  n=k;                                * n is used as the ID variable for the animation;
  if index<=k then output;            * Output retains the previous iterations;
 end;
 drop k;
run;

proc sort data=sim4;                  * Final sort prior to the animation call;
 by n index;
run;
```

To run simulations efficiently it is about changing your mindset moving away from using macro loops to iterate your data instead creating a matrix containing all the detail you need before passing to your final datastep or procedure.

ANIMATION IN SAS

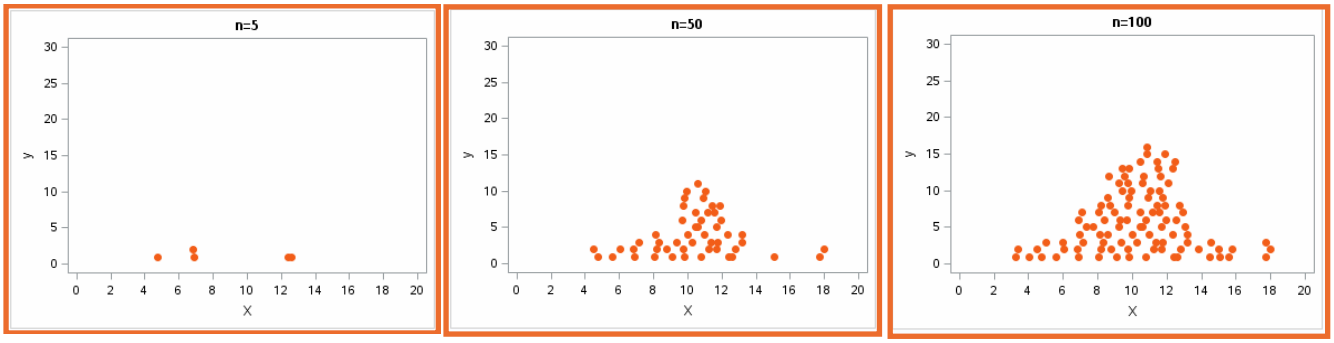
Plotting the unrounded normally distributed values against the frequency of the rounded x values enables us to avoid the columns of dots that the rounded values or a dot plot would give us. See the below for example of what we want to avoid: Note larger figure are available in the appendices



The animation code itself uses ods graphics to produce a gif file containing our animation.

```
ods graphics / imagefmt=GIF width=4in height=3in;     * each image is 4in x 3in GIF ;
options papersize=('4 in', '3 in')
        nodate nonumber
        animduration=0.5 animloop=yes noanimoverlay
        printerpath=gif animation=start;             * start recording images to GIF ;
ods printer file='** File Directory **/Anim.gif';     images saved into animated GIF ;

proc sgplot data=plot ;
   by n;
* each image is a time series ;
   scatter x=x y=y/ markerattrs=(symbol=circlefilled color='#F25D18' size=8);
   xaxis values=(0 to 20 by 2);
   yaxis values=(0 to 30 by 5);          * set common scales for all graphs ;
run;

options printerpath=gif animation=stop;                  * stop recording images ;
ods printer close;
```

As you can see below using the unrounded values plotted against the frequency variable Y produces a graph which, as the number of data points increases the animation starts to resemble a normal distribution. Note larger graphs are available to view in the appendices



SIMULATION IN R

As part of my investigation into simulating data I felt I needed to explore tools beyond SAS. R seems to be the programming language of choice for simulating data in data science. R is an object orientated programming language whose data forms include vectors, matrixes and data frame (Similar to datasets in SAS) as R uses vectors and matrixes as standard data forms it lends itself to simulating data efficiently. The Base R language is extended with user written packages and we start our R script by calling those that we require for the script.

```
## Required packages
library(tidyverse)
library(tweenr)
library(gganimate)
library(gifski)
library(ggplot2)
library(extraDistr)
```

Like the proc IML SAS code we start our simulation by creating three vectors containing: the number of observations per frame; a vector of ID values from 1 to n; our Normally distributed random values with mean 10 and standard deviation 3

```
obs_per_state <-  seq(5,100, by = 5) ## vector of observations per frame
max_obs <- seq(1,100)  ## number of observations last frame
randomData1<- rnorm(100, 10, 3) ## vector of simulated random values
```

Using the vector containing the number of observations per frame at the ith position we populate the two lists the first containing i random values and the second containing 1 to i values. For those not familiar with lists in R they are a data form which can contain vectors but also have a position which we use to have i values at position i forming the iterative values we require.

```
## Creation of 2 lists: observations and per frame and id of each observation
tu_list <- list()
tu_list_2 <- list()

for (i in obs_per_state) {
  tu_list[[i]] <- randomData1[1:i]
  tu_list_2[[i]]<- max_obs[1:i]}
```

We then unpack these lists into two vectors containing our random values and the corresponding ID value. The first 65 observations of the vectors we have now formed are capture below.

```
## Turning list into vectors to be used as columns in the animation dataset
tu_list_vec <- unlist(tu_list)
tu_list_vec_2 <- unlist(tu_list_2)
```

```
> tu_list_vec
  [1] 16.761689  8.865178 12.358355 10.312313 13.571565 16.761689  8.865178 12.358355
  [9] 10.312313 13.571565 12.932158  9.113887  9.794391  3.078726  8.866470 16.761689
 [17]  8.865178 12.358355 10.312313 13.571565 12.932158  9.113887  9.794391  3.078726
 [25]  8.866470  8.680197 11.175654 12.859231 10.635668  7.826560 16.761689  8.865178
 [33] 12.358355 10.312313 13.571565 12.932158  9.113887  9.794391  3.078726  8.866470
 [41]  8.680197 11.175654 12.859231 10.635668  7.826560  9.789240  8.341111  3.824764
 [49]  9.760964 10.078544 16.761689  8.865178 12.358355 10.312313 13.571565 12.932158
 [57]  9.113887  9.794391  3.078726  8.866470  8.680197 11.175654 12.859231 10.635668
```

```
> tu_list_vec_2
  [1]  1  2  3  4  5  1  2  3  4  5  6  7  8  9 10  1  2  3  4  5  6  7  8  9 10 11 12 13
 [29] 14 15  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20  1  2  3  4  5  6
```

We use tibble function, part of the tidyverse package, to create our simulated dataset. We also round our random values to use in the Y derivation and create our 'n' value which contains the number of observations per frame.

```
## Dataset for the animation
sim_Data1 <- tibble(
  ID = tu_list_vec_2,
  X = tu_list_vec,
  XROUND = round(tu_list_vec, digits = 0), ## rounds random values
  n = as.factor(rep(obs_per_state, obs_per_state)))
```

	ID	X	XROUND	n
1	1	9.840833	10	5
2	2	8.663836	9	5
3	3	14.386658	14	5
4	4	11.111366	11	5
5	5	14.799372	15	5
6	1	9.840833	10	10
7	2	8.663836	9	10
8	3	14.386658	14	10
9	4	11.111366	11	10
10	5	14.799372	15	10
11	6	15.162625	15	10
12	7	5.817150	6	10
13	8	10.526227	11	10
14	9	17.054662	17	10
15	10	5.468197	5	10

Lastly, we produce our Y value by grouping the XROUND column by n and then calculating the frequency of XROUND values within each group.

```
## This bit computes the cumulative frequency for each rounded value within each frame
```

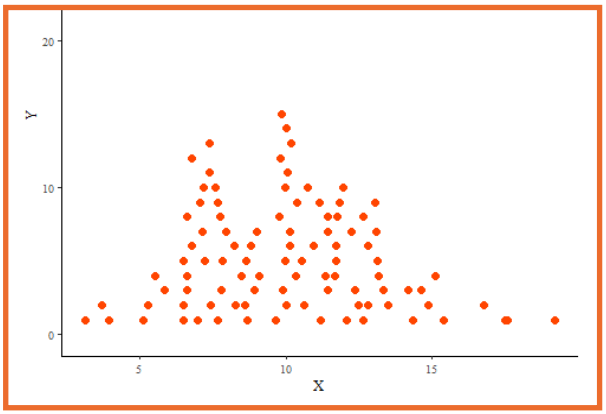
```
sim_Data2 <- sim_Data1%>%
  group_by(n) %>%
  mutate(y_Frequency = sapply(1:length(XROUND),
                              function(x)sum(XROUND[1:x] == XROUND[x])))

sim_Data3 <- select(sim_Data2, ID, X, XROUND, y_Frequency, n ) # reordering vars
```

	ID	X	XROUND	y_Frequency	n
1	1	9.840833	10	1	5
2	2	8.663836	9	1	5
3	3	14.386658	14	1	5
4	4	11.111366	11	1	5
5	5	14.799372	15	1	5
6	1	9.840833	10	1	10
7	2	8.663836	9	1	10
8	3	14.386658	14	1	10
9	4	11.111366	11	1	10
10	5	14.799372	15	1	10
11	6	15.162625	15	2	10
12	7	5.817150	6	1	10
13	8	10.526227	11	2	10
14	9	17.054662	17	1	10
15	10	5.468197	5	1	10

ANIMATION IN R

Creating the Animation in R is relatively straight forward once you have the right packages installed. Here we use the gifski package we can see the code efficiency of R when we do the comparison between code required in R and SAS, R only requires one line of code as opposed to 11 lines of code in SAS.

```
## animation
ggplot(sim_Data3, aes( x=X, y=y_Frequency)) +
  geom_point( size = 3, color = 'orangered1') +
  transition_states( n, transition_length = 1, state_length = 5)
```



RUNTIME COMPARISION

Both SAS and R have ways of profiling your code. Using the SAS code, we can time both the simulation and also different sections of our code to identify where the simulation is running inefficiently. Using this code I also timed how long the my original simulation ran to compare it against the more efficient code to measure the improvement in time taken. I ran the efficient SAS code twice, once with a by value of 1 so that 1000 frames are produced; then again with a by value of 5 so that 200 frames are produced. I believe this gives a fairer comparison against the original SAS code.

```
%let _timer_start = %sysfunc(datetime());
...
...
...
data _null_;
 dur = datetime() - &_timer_start;
 put 30*'-' / 'Total Duration :' dur time13.2 / 30*'-';
run;
```

The table below shows the significant efficiency gained. The efficient SAS code runs over 180 times faster than the original code. Interestingly, increasing the by value to 5 has a minute impact on the efficiency, but note the resulting animation will be produced faster as it will have 1/5 of the frames. In fact when the n value is increased to 10,000 the simulation only took 5.06 seconds to run.

I was slightly surprised by the time taken to run the R simulation which is 15 times slower for 1000 observations. But there could be mitigating factors both SAS and R are server based with similar capabilities but there is no way of knowing server load at time of run. Perhaps the R script is not the most efficient script for example the data manipulation section, in particular the derivation of the y_frequency variable, perhaps this is an area where the SAS code has advantages over the R script. I feel that certainly efficiencies could be found within the current R script to make is as efficient as the SAS code.

n=1000	Original Inefficient SAS Code	Efficient SAS Code with By Value 1	Efficient SAS Code with By Value 5	R Code with observations per frame 5
Time (seconds)	38.16	0.22	0.21	3.21

CONCLUSION

Simulating data is potentially not an everyday task for clinical programmers, but the skills I have learned and developed have applications in all programming tasks where we are dealing with large amounts of data.

- One of the main skills is recognising what causes program to run slowly to be able to identify code which has a high overhead but is performing little work.
- Simulating data efficiently requires a change in mindset from dynamically creating your data observation-by-observation to instead building the vectors and matrices that you require and using them to create your required data with as few steps as possible.
- What I have also found is that SAS can perform efficiently, I think we need to be careful with the narrative that R is always more efficient than SAS. I believe it very much depends on the task at hand and it is about picking the right tool for the job.
- Having said that I do not think the R script I put together for this paper is as efficient as it could be and further research could be done to perform the simulation more efficiently using R.

- Working with R, one area where it is obviously more efficient is code efficiency. Only a relatively small amount of code is required to achieve the program aims.
- Animations are a great way to bring data to life and are a good option for training materials.

REFERENCES

Wicklin, Rick. 2012. 'Eight tips to make your simulation run faster'.
[ONLINE] Available at: https://blogs.sas.com/content/iml/2012/06/06/tips-to-make-your-simulation-run-faster.html. [Accessed 29 September 2019].
Wicklin, Rick. 2012. Simulation in SAS: The slow way or the BY way.
[ONLINE] Available at: https://blogs.sas.com/content/iml/2012/07/18/simulation-in-sas-the-slow-way-or-the-by-way.html. [Accessed 29 September 2019].
Pratt, Jesse. 2016. Using Animation to Make Statistical Graphics Come to Life.
[ONLINE] Available at: https://support.sas.com/resources/papers/proceedings16/10920-2016.pdf. [Accessed 29 September 2019].

ACKNOWLEDGMENTS

A special thank you to Sam Munoz Vicente (Statistician, GSK) for all of his assistance putting the R simulation together and answering my very basic questions on how R simulates data.

CONTACT INFORMATION

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

Ben White
GlaxoSmithKline
Email: ben.e.white@gsk.com

APPENDIX 1

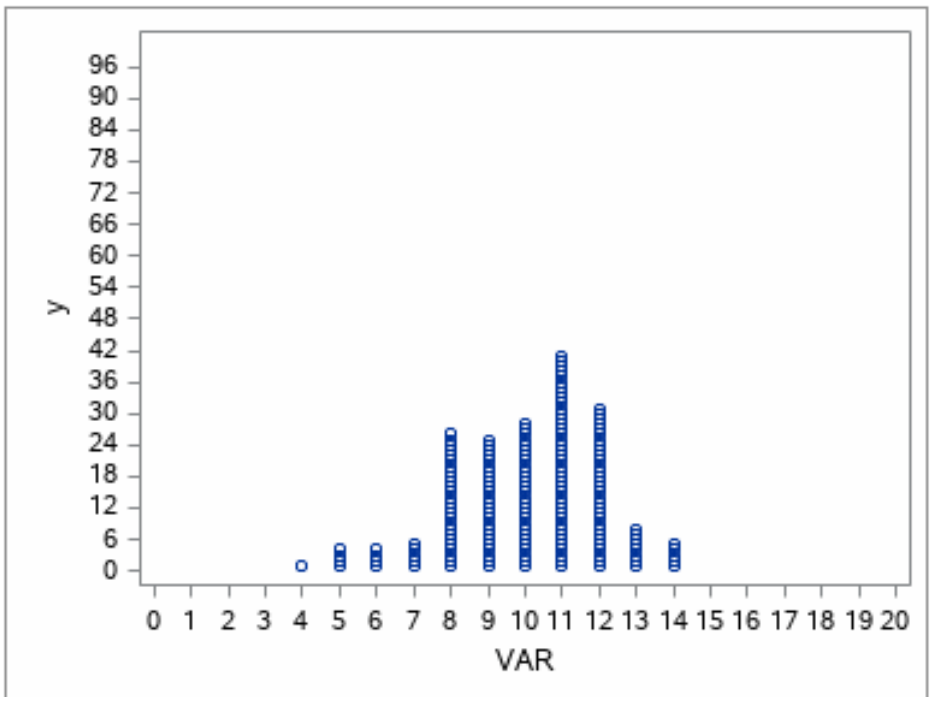
```
%let dset=test;
%let n=100;

data &dset.1;
 input ID $ COUNT;
 datalines;
A1 1
run;

%macro Jloop(n=,dset=);

data &dset.1;
 set &dset.1;
 VAR=round(rand('normal',10,2),0.1);
 y=1;
run;

%do i=2 %to &n;

data &dset.&i.;
 set &dset.%eval(&i.-1);
 where COUNT=&i.-1;
run;

proc sql;
 insert into &dset.&i. (ID,COUNT,y) values ("A&i.",&i.,1);
quit;

data _null_;
 set &dset.&i.;
 call symput('newvar',round(rand('normal',10,2),0.1));
run;

data &dset.&i.;
 set &dset.&i.;
 COUNT=&i;
 VARCOMP=&newvar;
 if ID="A&i" then VAR=&newvar;
 else if VARCOMP=VAR then y=y+1;
 drop VARCOMP;
run;

proc append base=&dset.&i. data=&dset.%eval(&i.-1);
run;
%end;

proc datasets lib=work;
 save &dset.&n.;
run;

proc sort data=&dset.&n.;
 by COUNT ID;
run;

%mend;

%Jloop(n=&n,dset=&dset);
```

APPENDIX 2

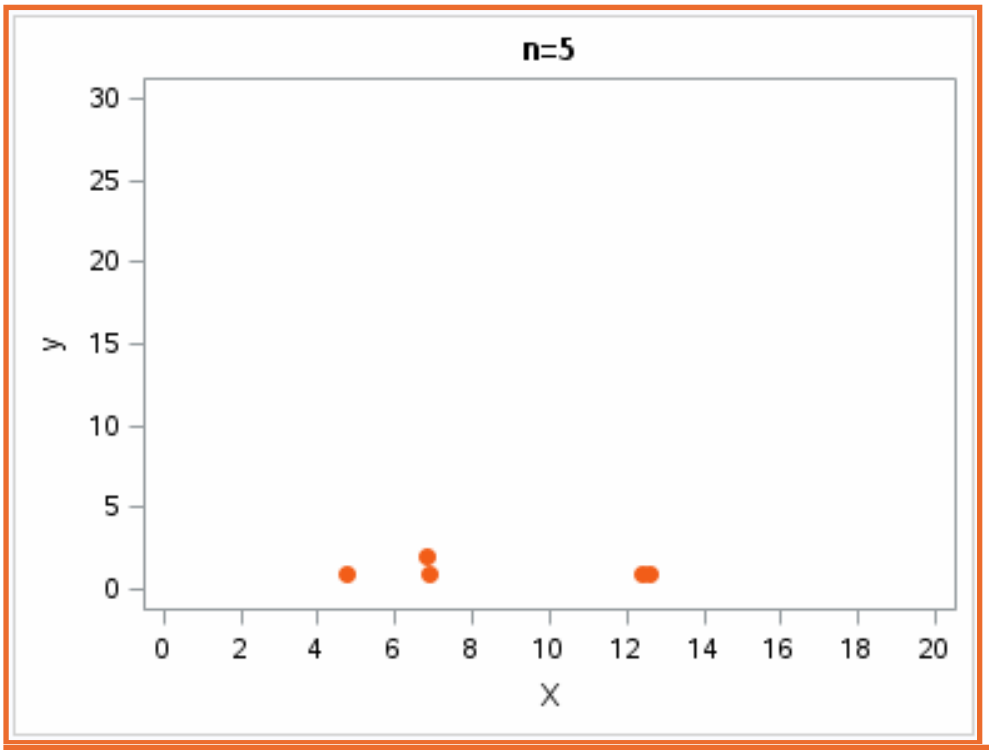
```
%let n=10000;                              * n is the number of data points;
%let _timer_start = %sysfunc(datetime());
proc iml;
 names={index x Xround};                 * Naming of variables in the sim1 dataset;
 i = T(1:&n);                            * Creation of a i index matrix;
 X1 = j(&n,1);                           * Initialise 1 by n marix;
 call randseed(1234);                    * Ensure the same values are produced each run;
 call randgen(X1,'normal',10,3);         * Generate n norm values with mean=10 and sd=3 ;
 XR=round(X1,1);                         * Round normally distributed data;
 X2=insert(i,X1,0,2);                    * Combine i and X1 matrices;
 X =insert(X2,XR,0,3);                   * Insert rounded values also ;
 create sim1 from X [colname=names];     * Create sim1 dataset;
  append from X;
 close sim1;
quit;

proc sort data=sim1 out=sim2;            * Sort prior to Y calculation;
 by Xround index;
run;

data sim3;                               * Datastep to calculate the Y value;
 set sim2;
 by Xround;                              * Using a by statement avoids a loop;
 retain y;
 if first.Xround then Y=1;               * Assign the number of values which fall into
the;
 else Y=Y+1;                             * Xround category for ascending index;
run;

data sim4;                               * Create the final dataset for the animation;
 set sim3;
 do k=5 to &n by 5;                      * Each loop adds an additional 5 values;
  n=k;                                   * n is used as the ID variable for the animation;
  if index<=k then output;               * Output retains the previous iterations;
 end;
 drop k;
run;

proc sort data=sim4;                     * Final sort prior to the animation call;
 by n index;
run;

data _null_;
 dur = datetime() - &_timer_start;
 put 30*'-' / 'Total Duration :' dur time13.2 / 30*'-';
run;

data plot;
      set sim4;
run;

ods graphics / imagefmt=GIF width=4in height=3in;     * each image is 4in x 3in GIF ;
options papersize=('4 in', '3 in')
        nodate nonumber
        animduration=0.5 animloop=yes noanimoverlay
        printerpath=gif animation=start;             * start recording images to GIF ;
ods printer file='**Folder path**Anim.gif';  * images saved into animated GIF ;
```

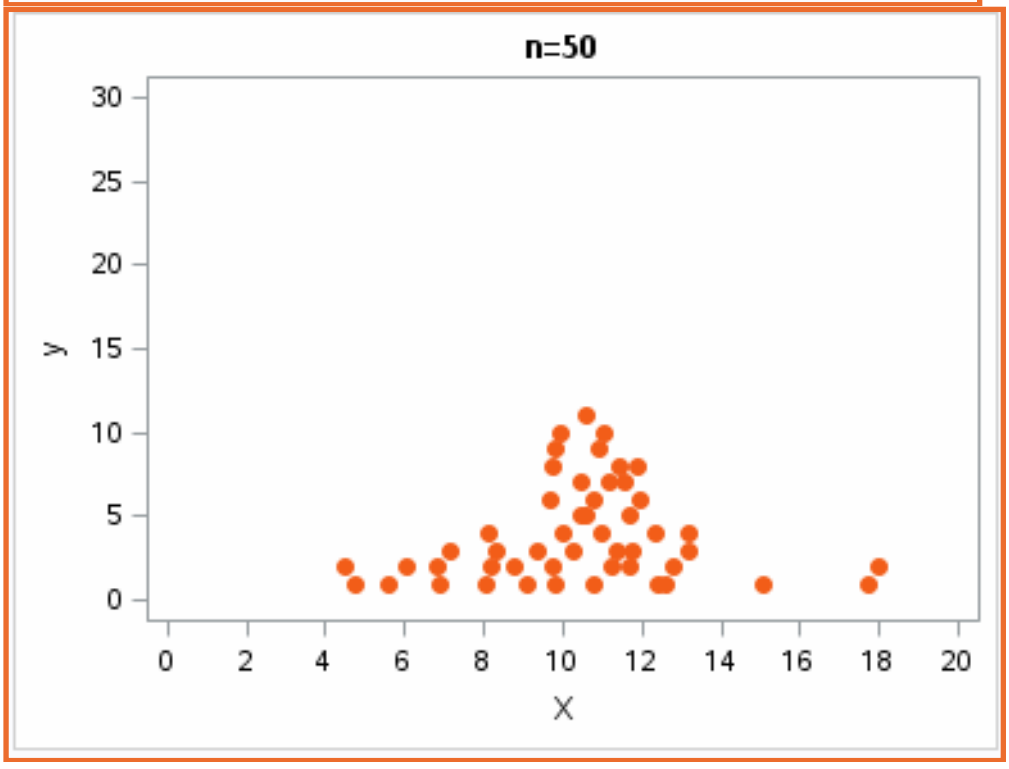
```
proc sgplot data=plot ;
   by n;
   scatter x=x y=y/ markerattrs=(symbol=circlefilled color='#F25D18' size=8);
* each image is a time series ;
   xaxis values=(0 to 20 by 2);
   yaxis values=(0 to 30 by 5);                  common vertical scale for all graphs ;
run;

options printerpath=gif animation=stop;               * stop recording images ;
ods printer close;
```

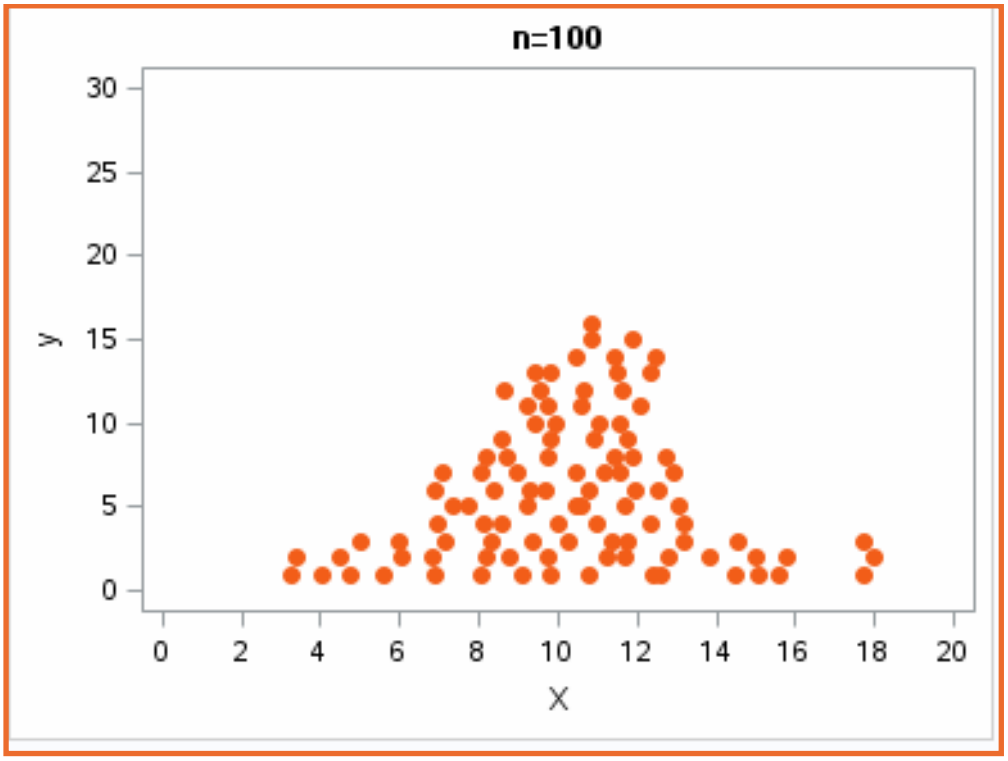
APPENDIX 3



APPENDIX 4


n=5
30
25
20
15
10
5
0
y
0
2
4
6
8
10
12
14
16
18
20
X


n=50
30
25
20
15
10
5
0
y
0
2
4
6
8
10
12
14
16
18
20
X


n=100
30
25
20
15
10
5
0
y
0
2
4
6
8
10
12
14
16
18
20
X