PhUSE EU Connect 2018

Paper CT04

The Data Step Crystal Ball: Observing Future Rows

Michael Allie, Biometrics (Programming), IMED Biotech Unit, Early Clinical Development, AstraZeneca, Cambridge, United Kingdom

ABSTRACT

A SAS® observation is often thought to be equivalent to a row of data. What if it was possible to simultaneously "observe" multiple rows within a data step? There is increasing benefit to utilising information from multiple events when analysing clinical trial data; whether deriving timing variables or visit responses, it is advantageous to consider not just a single event but those that occurred before and/or after. Events are typically captured across different rows and transposing is not always a feasible solution. While tools such as the retain statement and the lag operator incorporate previous values, reading ahead is more difficult and procedures which utilise entire data columns lack the flexibility to create bespoke algorithms. This paper describes and compares a selection of novel programming techniques including reflexive merging and hash iterator objects to construct the data step crystal ball, a tool for observing beyond the current row of data.

INTRODUCTION

The initial problem which sparked the inspiration for this paper seems simple on first impression; I had to create a flag that indicated when a specific criterion was met for two or more consecutive records.

For this paper, we will flag instances when we see two or more consecutive lab results (LBSTRESN) that are lower than the standard lower normal limit (LBSTNRLO) within the same lab test code (LBTESTCD) for a given subject (SUBJECT). Below is an example of the required result in some simulated data.

	SUBJECT	LBTESTCD	LBSTRESN	LBSTNRLO	consecutiveLowFlag
1	1	GLUC	6.7630592416	3.8857	
2	1	GLUC	3.3916498243	3.8857	Y
3	1	GLUC	3.5584941976	3.8857	Y
4	1	GLUC	2.5299413569	3.8857	Y
5	1	GLUC	4.7909099201	3.8857	
6	1	GLUC	5.2967146997	3.8857	
7	1	GLUC	6.3083869808	3.8857	
8	1	GLUC	5.3322204975	3.8857	
9	1	GLUC	6.1226277479	3.8857	
10	1	GLUC	4.5676621504	3.8857	
11	1	GLUC	3.7288569046	3.8857	
12	1	GLUC	4.5463340997	3.8857	

Notice that record 2 must be flagged. This is the reason that some method of looking forward in the dataset is required.

Data steps naturally read records one at a time so when observation 2 is read in via the SET statement, the only way to know at that point that record 2 needs to be flagged would be to know that record 3 is also abnormally low. Unfortunately, when record 3 is read in there is no way to go back and edit record 2.

To solve this problem, I theorised a few potential solutions:

1. Make multiple run-throughs of the data utilising only the *lag* operator; once in the original order (which would correctly flag records 3 and 4 in our example), sort the data in reverse chronological order and then make another run-through using similar logic (which would flag record 2 as it would appear below record 3) and then finally returning the data to the original order afterwards.
2. Create a row number variable and an offset row number variable and reflexively merge the next row onto the original data
3. Load the data into a hash object and use a hash iterator to travel backwards and forwards through the data to determine when low results occurred consecutively

I have since revisited this problem with the intent of finding out which solution is the most efficient.

METHOD 1: USING LAG OPERATOR AND REVERSE-CHRONOLOGICAL SORT

We've already discussed how we can quite easily look backwards in a dataset via the *lag* operator; Sorting our data backwards allows us to use this same method to look "forwards". While logically quite simple, this approach requires the most run-throughs of the data: the initial data step, sorting the data in reverse, a 2nd data step and then finally sorting the data back into its original order.

DATA STEP CODE

```
  * Calling lag operator for all relevant variables;
  lagUsubjid = lag(usubjid);
  lagLbtestcd = lag(lbtestcd);
  lagLbstresn = lag(lbstresn);

  * Checking that current record is the same usubjid*lbtestcd group as previous;
  if usubjid = lagUsubjid and lbtestcd = lagLbtestcd then do;

    * Checking if current observation is abnormally low;
    if . < lbstresn < lbstnrlo then do;

      * Creating flag if previous record was also abnormally low;
      if . < lagLbstresn < lbstnrlo then consecutiveLowFlag = "Y";
    end;
  end;

run;
```

It is important to invoke the *lag* operator for every observation as the value of lag(variable) relates to the last time the operator was used, *not* necessarily the value from the previous observation. We also see here that a check is required to ensure that values from the previous record belong to the same subject*test group.

The dataset can then be sorted in reverse order and similar logic applied to observe the (chronologically) future rows, before finally sorting the data back into chronological order to produce the required result.

METHOD 2: REFLEXIVE OFFSET MERGE COMBINED WITH LAG OPERATOR

Another approach to looking ahead is to merge the next row onto the current one. This can then be used in conjunction with the lag operator to look both forwards and backwards through the dataset.

The entire process is accomplished through two data steps. First, the automatic variable _N_ is used to create a variable with the current row number and a second variable that is one less than the row number.

```
data labRowData;
  set originalLab;
  currentRow = _n_;
  mergeIndex = _n_ - 1;
run;
```

In the second and final step, the merge index variable is renamed to reflexively merge the data set with a one row offset version of itself. Other variables from the offset dataset are renamed to avoid overwriting the values in the original dataset.

```
* Merging next row onto current row;
data finalLabDataMethod2 (drop=future: lag:) ;
  merge labRowData (drop=mergeIndex in=a)
        labRowData (keep = usubjid lbtestcd lbstresn mergeIndex
                    rename = (usubjid=futureUsubjid lbtestcd=futureLbtestcd
                              lbstresn=futureLbstresn mergeIndex=currentRow)
                       );
  by currentRow;
```

```
if a;
```

The lag operator is then used similarly to method 1 to look backwards; reading future values is accomplished by utilising the future: variables from the offset dataset.

```
* Checking if the consecutive flag has already been set for this observation;
if consecutiveLowFlag = "" then do;

  * Ensuring that current observation is in the usubjid*lbtestcd group as future;
  if usubjid = futureUsubjid and lbtestcd = futureLbtestcd then do;

    * Assigning flag if future observation is also abnormally low;
    if futureLbstresn < lbstnrlo then consecutiveLowFlag = "Y";
  end;
end;
```

The main difference here is that checks in both directions can be performed at once. Using the lag*:* variables to look backwards and the future: variables to look ahead. This solves the problem in two run-throughs of the data.

METHOD 3: STATIC HASH OBJECT

The final method I initially devised was to load all the observations into a hash object and use a hash iterator to travel backwards and forwards through the hash object; this would act as a “crystal ball” that could observe any row in the dataset and generate the required result in just a single run-through of the data.

WHAT IS A HASH OBJECT?

Functionally, a hash object is an additional dataset that exists solely in memory for the duration of a single data step. They are traditionally used to efficiently merge datasets together but there are other novel uses for them, such as the scenario described in this paper.

DEFINING A HASH OBJECT

There are a few steps involved in successfully constructing the data step hash object we need:

1. Begin the data step normally

2. Explicitly initialise a set of variables that will access the hash object data into the PDV

   This must be done before the actual hash object is defined. These are essentially placeholder variables for the data from the hash object. The data step will cease execution with an error if there is no location in the PDV explicitly initialised prior to defining the hash object.

3. Define the data that the hash object will contain

   The same lab dataset is loaded into the hash object, keeping only a subset of the variables; this reduces the amount of work involved in initialising variables within the data step and minimises the amount of memory the hash object will occupy. These variables are also renamed into new variables with a dynamic: prefix to differentiate them from the variables belonging to the main dataset. If this is not done, then the data from the hash object would overwrite the main dataset variables with the same name (similar to what can happen with a data step match merge)

   - The dcl hash (short for declare hash) statement is used to specify the dataset that will be loaded into the hash object (with dataset options) and that the data is ordered in ascending order of the key variables
   - The defineKey method is used to choose the key variables. These are used to determine the order of the records in the hash object. This is why the date variable dynamicLbdtc is included; it ensures the hash object data is sorted in chronological order but does not need to be explicitly used in any other statements.
   - The defineData method determines which variables will have their data loaded into the PDV
   - The defineDone method signals the end of the hash object definition

4. Define a hash iterator that will be used to cycle through the data in the hash object

   The hash iterator is used to point to a specific item within the hash object and load that item's data into the variables initialised for the hash object.

1
```
data finalLabDataMethod3;
  set originalLab;
```

2
```
* Initialising all hash object variables to PDV;
length dynamicUsubjid dynamicLbtestcd dynamicLbdtc $ 50 dynamicLbstresn 8.;
call missing(dynamicUsubjid,dynamicLbtestcd,dynamicLbdtc,dynamicLbstresn);
```

```
* Creating hash object;
if _n_ = 1 then do;
```

3
```
dcl hash crystalBall(dataset:catt("originalLab (keep = usubjid lbtestcd lbstresn
                                                     lbdtc"
                                  ," rename = (usubjid=dynamicUsubjid
                                               lbtestcd=dynamicLbtestcd"
                                  ," lbstresn=dynamicLbstresn
                                     lbdtc=dynamicLbdtc))"
                                )
                    ,ordered:"a"
                   );
  crystalBall.defineKey("dynamicUsubjid","dynamicLbtestcd","dynamicLbdtc");
  crystalBall.defineData(all:"yes");
crystalBall.defineDone();
```

4
```
* Creating hash iterator object to traverse multiple observations;
dcl hiter labTraverse("crystalBall");
```

```
end;
```

It is important that this hash object is only defined once; this is why the declaration is contained within a do block for the first record (_n_ = 1). Data from the hash object is loaded into active memory (as opposed to normal data step processing which reads from disk and only loads data from a single row at a time into memory) and defining multiple hash objects will unnecessarily use up a large amount of data and eventually cause the data step to stop executing. Once the hash object and hash iterator have been created, they can then be used to access data from any record in the dataset.

The hash iterator .next() method loads the next item from the hash object into memory, or the first item if neither this method nor the .first() method have been called. A numeric variable positionTracker is created, containing the relative position of the hash object record compared to the main (read in through the set statement) record.

```
* Advancing hash object forward one observation to retrieve data for main row
  (+0);
rc = labTraverse.next();
positionTracker = 0;


* Checking if current record is abnormally low;
if lbstresn < lbstnrlo then do;
```

The .prev() method is then used to check the observation from the previous record rather than the lag operator which was used in the previous methods. For all hash iterator methods, a return code (rc) of 0 indicates that the operation was successful and a non-zero return code indicates failure. If the .next() method returns a non-zero return code, that normally indicates that the end of the hash object has been reached; in that instance the explicitly initialised hash object variables would not be updated and the values for the last item in the object will remain. The rc is checked to make sure that the previous record was successfully read in so the positionTracker variable remains accurate.

```
* Load in previous record (-1);
rc = labTraverse.prev();

if rc = 0 then do;
  if usubjid=dynamicUsubjid and lbtestcd = dynamicLbtestcd
    and dynamicLbstresn < lbstnrlo then consecutiveLowFlag = "Y";
  positionTracker = -1;
end;

* Only check future observation if flag has not already been set based on
  previous record;
if consecutiveLowFlag ^= "Y" then do;
```

If looking ahead is required, this is accomplished by using the .next() method twice to advance two items ahead in the hash object.

```
  * Advance ahead 2 records to load in one record ahead of current main record
   (+1);
  do i = 1 to 2;
    rc = labTraverse.next();
    if rc = 0 then positionTracker+1;
  end;

   * Checking that next record has been loaded into hash object;
   if rc = 0 then do;

    * Checking that future row is within the same usubjid*lbtestcd group;
    if usubjid=dynamicUsubjid and lbtestcd = dynamicLbtestcd then do;
      if dynamicLbstresn < lbstnrlo then consecutiveLowFlag = "Y";
    end;
   end;

end;
```

Finally, the positionTracker variable is used to control calls to the .next() or .prev() methods to return to the current record's information in the hash object ready for the next iteration in the data step.

```
* Returning to current observation;
if sign(positionTracker) = -1 then do i = 1 to abs(positionTracker);
  rc = labTraverse.next();
end;
else if sign(positionTracker) = 1 then do i = 1 to positionTracker;
  rc = labTraverse.prev();
end;
```

THE MEMORY CONSTRAINT

In benchmarking the three methods discussed thus far, I noticed that the static hash object required large amounts of memory; this led me to adjusting the method and dynamically defining the hash object to partition the data into multiple subsets which are loaded into memory one at a time.

METHOD 4: DYNAMIC HASH OBJECT

Since all the data from the hash object is loaded into memory, defining hash objects from large datasets requires proportionally more memory. As mentioned previously, it is possible to exceed the memory limit when the datasets become extremely large.

A dynamically defined hash object resolves this memory limitation, but it can also negatively impact performance. Defining a hash object has an overhead in terms of processing; the data from the hash object must be read from the disk and into memory, adding to the execution time. It then becomes a balancing act of: avoiding defining hash objects that are too large because that will use a lot of memory and refraining from defining hash objects too frequently, as reading from the disk many times will slow processing to a crawl.

By utilising the automatic variable _n_ to dynamically set firstObs= and obs= dataset options, a hash object can be

created from a range of observations; in this example the dataset is partitioned into subsets of 50,000 records at a time which keeps memory usage relatively low and doesn't require reading from the disk an excessive number of times. Now, a hash object is not just defined at the first row but also whenever the flag newHashRequired is set to "Y".

```
 * Creating hash object of next 50000 records;
  if _n_ = 1 or newHashRequired = "Y" then do;

    dcl hash crystalBall(dataset:catt("originalLab (keep = usubjid lbtestcd lbstresn
                                                          lbdtc"
                                    ," rename = (usubjid=dynamicUsubjid
                                                 lbtestcd=dynamicLbtestcd"
                                    ," lbstresn=dynamicLbstresn lbdtc=dynamicLbdtc)"
                                    ," firstObs=",_n_," obs=",_n_ +50000,")"
                                    )
                        ,ordered:"a"
                       );
      crystalBall.defineKey("dynamicUsubjid","dynamicLbtestcd","dynamicLbdtc");
      crystalBall.defineData(all:"yes");
    crystalBall.defineDone();

    * Creating hash iterator object to traverse multiple observations;
    dcl hiter labTraverse("crystalBall");


    * Resetting flag requesting next hash object to be created;
    newHashRequired = "";

  end;
```

This adds to the complexity; if the end of a hash object occurs within a usubjid*lbtestcd group then looking forward would not be possible at that time because the next record would not be in the hash object.

```
* Checking that data for the future record was successfully loaded into PDV;
      if rc = 0 and newHashRequired="" then do;
        if usubjid = dynamicUsubjid and lbtestcd = dynamicLbtestcd
         and dynamicLbstresn < lbstnrlo then consecutiveLowFlag = "Y";
      end;
```

If a non-zero return code is returned from the .next() method in this instance then the end of the current hash object has been reached. When this occurs:

1. An additional hash object is created containing only the next record. (Note that dynamicLbdtc is not required here because there is only a single record in the hash object, so the order is irrelevant)
2. This data is loaded into the dynamic: variables via a new hash iterator object and then compared against the current record variables from the main dataset. Values are loaded into the same dynamic: variables initialised earlier. Once the variables are initialised to the PDV they can be overwritten as many times as required within the main data step iteration.
3. The hash object and iterator are deleted once the values from the hash object is no longer needed
4. The newHashRequired flag is set to "Y" which will cause the crystalBall hash object to be deleted and recreated with the next 50,000 records loaded in

```
      else do;

        dcl hash nextRow(dataset:catt("originalLab (keep = usubjid lbtestcd
                                                            lbstresn"
                                    ," rename = (usubjid=dynamicUsubjid
                                                 lbtestcd=dynamicLbtestcd"
                                    ,"           lbstresn=dynamicLbstresn)"
                                    ," firstObs=",_n_+ 1
                                    ," obs=",_n_ + 1,")"
                                    )
                        );
          nextRow.defineKey("dynamicUsubjid","dynamicLbtestcd");
          nextRow.defineData(all:"yes");
```

```
  nextRow.defineDone();

  dcl hiter nextRowIter("nextRow");
```

The .first() hash iterator method is utilised here. It simply retrieves the first item from the hash object into the dynamic: variables.

```
  * Loading in first record and checking for abnormality;
  rc = nextRowIter.first();
  if usubjid = dynamicUsubjid and lbtestcd = dynamicLbtestcd
   and dynamicLbstresn < lbstnrlo then consecutiveLowFlag = "Y";

  rc = nextRow.delete();
  rc = nextRowIter.delete();

  newHashRequired = "Y";

end;
```

Similar logic is applied when the current record in the main dataset relates to the first item in the hash object which prevents looking backwards via the hash object (the lag operator could be used here alternatively).

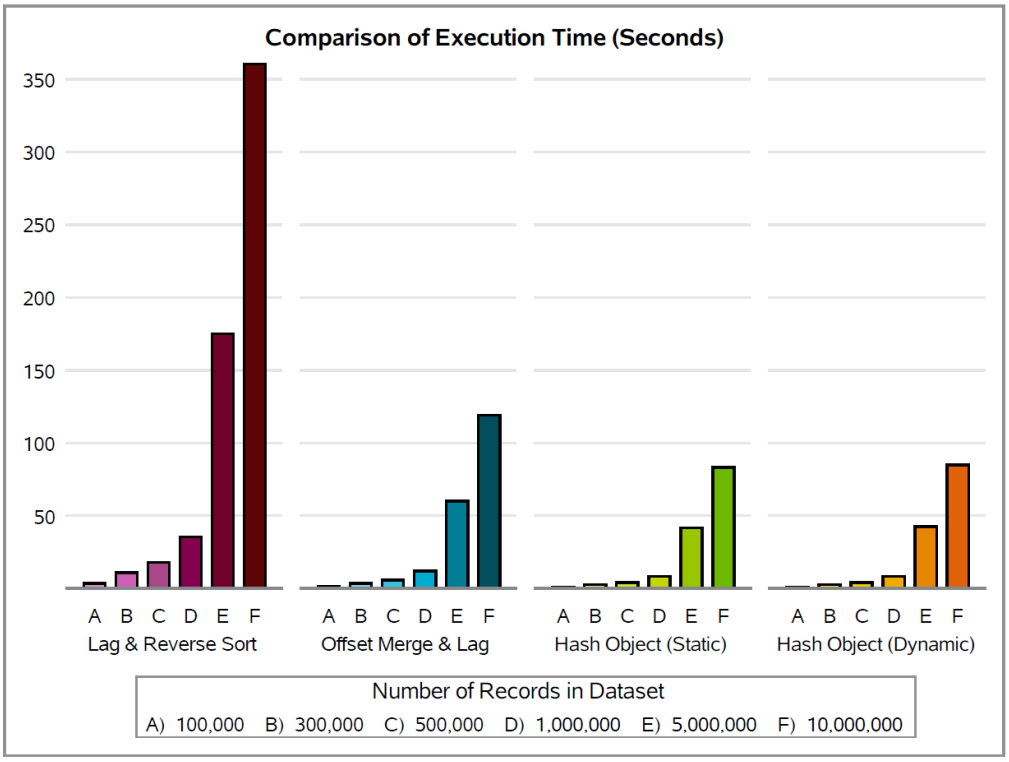
```
dcl hash previousRow(dataset:catt("originalLab (keep = usubjid lbtestcd
                                                       lbstresn"
                                  ," rename = (usubjid=dynamicUsubjid
                                               lbtestcd=dynamicLbtestcd"
                                  ,"           lbstresn=dynamicLbstresn)"
                                  ," firstObs=",_n_- 1
                                  ," obs=",_n_ - 1,")"
                                  )
                    );
  previousRow.defineKey("dynamicUsubjid","dynamicLbtestcd");
  previousRow.defineData(all:"yes");
previousRow.defineDone();

dcl hiter previousRowIter("previousRow");
```

COMPARISON OF THE METHODS

I ran each of the four methods described in this paper in turn on datasets of varying size:

- 100,000 records
- 300,000 records
- 500,000 records
- 1,000,000 records
- 5,000,000 records
- 10,000,000 records

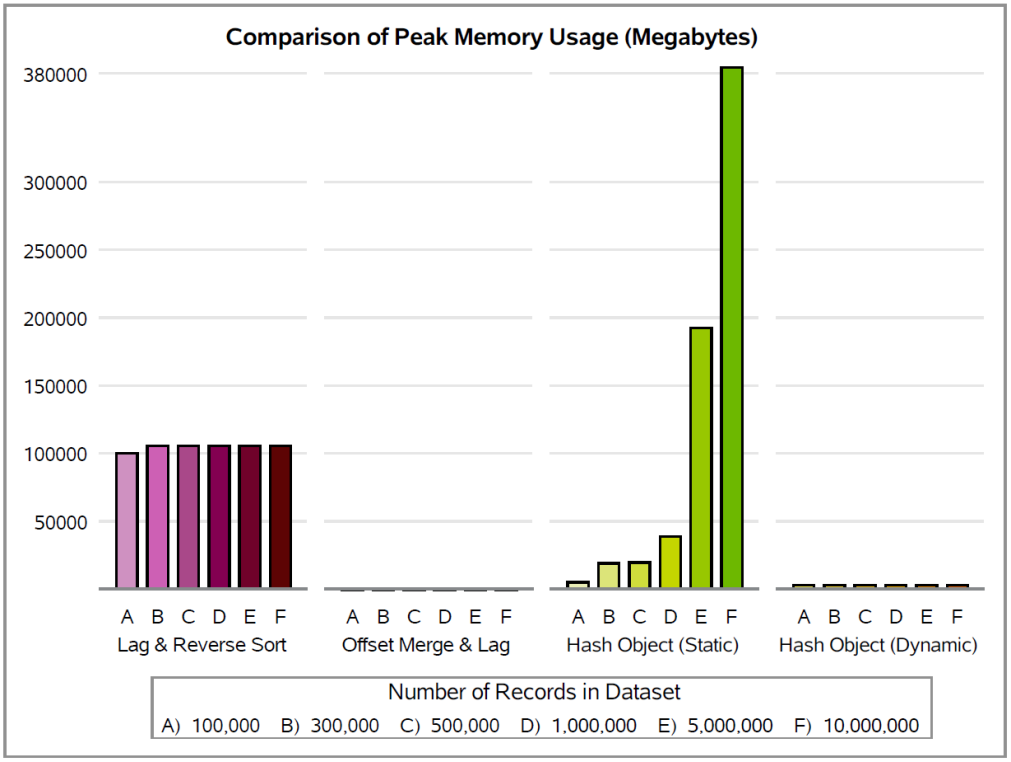
This process was repeated 20 times beginning at 12:00 midday and 20 times beginning at 12:00 midnight. All benchmarking information was then aggregated and averaged for each method. The runs during the night were consistently faster than the daytime runs but the difference was quite minor (1-2%). All statistics presented here are averaged between the daytime and nighttime benchmarking sessions. The code was run using SAS Enterprise Guide on a SAS Grid Linux environment. The exact runtimes may vary depending on the operating environment.



Sorting the data in reverse and using the lag operator was the slowest out of all the methods. This to be expected as it involves the most run-throughs of the data, including two sorts of the entire dataset which contribute the most to the total execution time.

The difference in the other three methods is much smaller though the offset merge method takes about 50% longer than the hash objects on average.

There is virtually no difference between the two hash object methods despite the dynamic hash object requiring reading from the disk more frequently.



The memory utilization from the static hash object increases proportionally to the size of the dataset.

Proc sort likely allocates a set amount of memory in advance which is independent to the size of the input dataset.

The low memory usage from the offset merge method is because data steps process directly from the disk and therefore only need to hold the data for a single observation in memory at a time

The consistent memory usage from the dynamic hash object was the main goal of partitioning the dataset in this manner and removing the limitation of the static hash object.

CONCLUSION

The Offset Merge and both Hash Object methods all run significantly faster than the Lag & Reverse Sort method (the baseline to be improved on). Within the three comparison methods, the hash objects are faster than the offset merge; though this efficiency does come at a cost in terms of the complexity of the code. The hash objects utilise syntax that is unlike most SAS statements and careful management of the PDV is required to achieve the intended result without

errors; it is also important to note that the static hash object method does not scale well with very large data as there is a limit to the amount of memory available which will cause the data step to fail when exceeded; I would always recommend the dynamic hash object method over the static hash object as this uses a fixed amount of memory which is irrespective of the size of the input dataset. Both hash object methods have the benefit of being able to look *n* rows ahead with ease. To observe multiple future rows using the offset merge method would require a new set of variables to be merged for each additional row which may impact the execution time and will add to the code required; the other methods have an easier time of this with lag2, lag3 etc. operators available for the Lag & Reverse Sort method and multiple calls to the .next() and .prev() hash iterator methods available to quickly traverse hash objects.

Overall, I would say that the offset merge method offers a good balance between simple syntax, respectable execution time and no concern over memory usage. Those that can handle the complexity of a dynamically declared hash object benefit from superior runtime and flexibility to observe multiple future rows without major updates to the code.

RECOMMENDED READING (HEADER 1)

Hash object syntax tip sheet: https://support.sas.com/rnd/base/datastep/dot/hash-tip-sheet.pdf

CONTACT INFORMATION

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

Michael Allie
AstraZeneca – Early Clinical Development
Da Vinci Building
Melbourne Science Park
Royston
SG8 6EH
Email: mike.ce.allie@gmail.com