Paper 062-2010

Challenges in Genomic Data Processing I — Multiple Small Files

Derek Morgan, Principle Solutions Group, St. Louis, MO
Bernard Omolo, Department of Biostatistics, University of North Carolina at Chapel Hill

ABSTRACT

Advances in the field of Genomics have made more data on the very building blocks of life available than ever before. When coupled with the burgeoning fields of Bioinformatics and Statistical Genomics, there is significant information that has been heretofore hidden in those simple chemical bonds. Unfortunately, bridging the divide between the fields of life sciences and information technology is not always straightforward. Data produced by genetic technology can be messy, and there is often a great deal of it. Despite the increases in computing power and natural language processing, order still must be brought to the chaos before it can be analyzed. This paper describes the development of an automated process to convert four gigabytes worth of tab-delimited data across multiple files into a SAS dataset.

THE PROBLEM

A collaborator brought in fifty-two tab-delimited files, each of which contained over 400,000 records (probes) with over one hundred fields. Each file represented one cell line, and contained gene expression data from each cell line. The straightforward approach of using Excel to view the data and the IMPORT wizard for conversion to SAS did not yield the desired result because the files were so large. Therefore, it was decided to use the old-fashioned INPUT statement to create usable SAS datasets. Each file had a lot of extraneous information at the top, and the order of fields in each file was not readily visible. In addition, the actual data did not necessarily start in the same place in each file, and each file used the same variable names, indicating specific statistics associated with each file. We wanted one dataset, with all the data, but the variable names would be constructed from the cell line identifier plus the statistic name. How could we do this without writing fifty-two different SAS programs with slightly different INPUT statements?

WHERE WE STARTED

The first thing we had to do was to break the problem into its component parts. We would need prefixes for the variable names, and these prefixes were based on information stored in one of the project files. We turned this into a SAS dataset for use in the program. There is some manipulation of the data to make sure that the prefixes would not cause invalid SAS names to be created (❶).

```
1   /* Read cell line FILE to get variable name prefixes */
2
3   DATA cellines;
4   LENGTH samp 3 celline $ 16 array_barcode $ 20;
5   INFILE "f:\CellLineKey.csv" PAD MISSOVER DLM="," FIRSTOBS=2;    ❶
6   INPUT samp celline $ array_barcode $;
7   celline = TRANSLATE(TRIM(celline),'_',' '); /* Turn spaces to underscores */
8   celline = TRANSLATE(celline,'_','('); /* Turn left parentheses to underscores */
9   celline = TRANSLATE(celline,'_',')'); /* Turn right parentheses to underscores */
10  celline = TRANSLATE(celline,'_','-'); /* Turn dashes to underscores */
11  IF SUBSTR(celline,LENGTH(celline),1) EQ '_' THEN
12     celline = SUBSTR(celline,1,LENGTH(celline)-1); /* remove trailing underscores */
13  IF samp NE . THEN
14     OUTPUT;
15  DROP samp;
16  RUN;
```

The next step is to create a dataset of the fully-qualified file names by getting a directory listing. Under a Windows-based system, this means that the path will need to be prepended to the file name from the ***dir*** command (line 18). You can use the ***ls*** command in LINUX/UNIX, and execute it so that the path name will be included in the file name itself. Since we need the information from the cell lines to create variable names along with the file names to read the ASCII files themselves, we put the two of them together.

```
17  OPTIONS NOXWAIT;
18  X "dir /b f:\bernard\*.txt > cells.txt"; /* create text FILE list of files */
19
20  /* Read list of files and turn it into a DATASET */
21  DATA files;
22  INFILE "cells.txt" PAD MISSOVER;
23  LENGTH fn $ 80 array_barcode $ 20;                    ❷
24  INPUT fn $ 1-80;
25  fn = CATS("f:\bernard\",fn); /* Add path to FILE names */
26  array_barcode = COMPRESS(CATS(SCAN(fn,2,'_'),'-',SCAN(fn,7,'_'),'-
    ',SCAN(fn,8,'_')),'.txt'); /* create barcode from filename */
27  CALL SYMPUT('nfiles',_n_); /* Put number of files into macro space */
28  RUN;
29  /* Add cell line names to FILE DATASET */
30  DATA files;
31  MERGE cellines files;
32  BY array_barcode;
33  RUN;
```

We don't know the variables to be created, so we do not know the variable types that we will be reading. Fortunately, this information is stored in line 9 of each file. We only process line 9 with the FIRSTOBS= and OBS= options on the INFILE statement (❸). The FILEVAR= option allows us to process multiple files based on the value of the *fn* variable from the *files* dataset. We read the line as one long character string, and we parse out the variable types with a second DATA STEP (❹).

```
34  /* Read variable types from files */
35  DATA test;
36  SET files;
37  LENGTH bigline $ 8192;
38  INFILE in FILEVAR=fn END=eof PAD MISSOVER LRECL=8192 FIRSTOBS=9 OBS=9;   ❸
39  DO UNTIL(eof);
40      INPUT bigline $ 1-8192;
41  OUTPUT;
42  END;
43  RUN;
44
45  /* Parse variable types */
46  DATA test2;
47  LENGTH type $ 80;          ❹
48  SET test;
49  i = 1;
50  DO WHILE(SCAN(bigline,i,'09'x) NE ' ');
51     type = SCAN(bigline,i,'09'x);
52     i + 1;
53     OUTPUT;
54  END;
55  RUN;
```

Now that we have file names, the prefixes for the variable names, and the variable types for each variable to be created, we can get the original variable names from the file. Once again, they've made the task possible by putting the variable names in line 10 of each file. The first DATA step reads the line with the variable names from each file, and the second DATA step creates the actual variable names we are going to use with the cell line prepended to each of the standard file names in the file. We sort them and merge them with the variable type dataset we created so that we have a type for each variable name.

```
56  /* Read variable names from files */
57  DATA prevars;
58  SET files;
59  LENGTH bigline $ 8192;
60  INFILE in FILEVAR=fn END=eof PAD MISSOVER LRECL=8192 FIRSTOBS=10 OBS=10;
61  DO UNTIL(eof);
62      INPUT bigline $ 1-8192;
63      OUTPUT;
64  END;
65  RUN;
66
67  /* Parse variable name lines into a list of variable names */
68  DATA test3;
69  LENGTH longname $ 80 varname $ 32;
70  SET prevars;
71  i = 1;
72  DO WHILE(SCAN(bigline,i,'09'x) NE ' ');
73
74  /* Add cell line to variable names */
75  /* Replace leading digit of cell line with an underscore */
76     IF anydigit(celline) EQ 1 THEN
77        longname = CATS('_',celline,'_',SCAN(bigline,i,'09'x));
78     ELSE
79        longname = CATS(celline,'_',SCAN(bigline,i,'09'x));
80     varname = SUBSTR(longname,1,32);
81     i + 1;
82     /* remove underscores for testing if key variable */
83     chk = STRIP(TRANWRD(varname,CATS(celline,'_'),''));
84     IF chk EQ 'GeneName' THEN
85        varname = "genename";
86     IF chk EQ 'ProbeUID' THEN
87       varname = "ProbeUID";
88    OUTPUT;
89  END;
90  DROP chk;
91  RUN;
92
93  PROC SORT DATA=test2;
94  BY array_barcode i;
95  PROC SORT DATA=test3;
96  BY array_barcode i;
97  RUN;
98
99  /* Create variable list dataset */
100 DATA varlist;
101 MERGE test2 test3;
102 BY array_barcode i;
103 DROP bigline;
104 RUN;
105
```

```
106 PROC SORT DATA=varlist ;
107 BY array_barcode i;
108 RUN;
```

Now for the macro. We knew in advance that the system producing the files creates them with 112 variables per file. Therefore, we can calculate which chunk of the variable name/type dataset (*varlist*) we need to process for each file. The first file will use records 1 through 112, the second, 113 through 224, and so on. We use a DATA _NULL_ step to write the INPUT statement that will process each file, based on the records processed from the *varlist* dataset. The DATA step indicated by ❺ writes the INPUT statement. It is written to a file that will be %INCLUDEd to facilitate debugging. There are more elegant ways to do this; indeed, I've created variable lists on the fly for statements and stored them in macro variables, but in this case, if something goes wrong, you cannot alter the INPUT statement without altering the programming that created the macro variable. It makes debugging easier since you can look at the code that is being used after the program has run.

```
109 *OPTIONS SYMBOLGEN MLOGIC MPRINT; /* debugging */
110 %MACRO doit;
111 %DO i=1 %TO &nfiles; /* Do this once for each file */
112
113 /* There are 112 variables in each file - only read variables for each individual
    file */
114 %LET FIRSTOBS = %EVAL((112*(&i-1))+1);
115 %LET obs = %EVAL(&FIRSTOBS+111);
116
117 /* Write INPUT statement to an ASCII file */                    ❺
118 DATA _null_;
119 SET varlist (FIRSTOBS=&firstobs OBS=&obs) END=eof;
120 FILE "f:\bernard\inputline&i..sas" NOPRINT LRECL=8192;
121 chk = STRIP(TRANWRD(varname,CATS(celline,'_'),''));
122
123 /* Put variables to keep into macro variables for later use */
124 IF chk EQ 'LogRatio' THEN
125    CALL SYMPUT('cv1',varname);
126 IF chk EQ 'PValueLogRatio' THEN
127    CALL SYMPUT('cv2',varname);
128
129 /* Write INPUT statement */
130 IF _n_ EQ 1 THEN DO;
131    PUT "LENGTH genename $ 100;";
132    PUT "INPUT " @;
133 END;
134 PUT varname +1 @;
135 IF type EQ 'text' or type EQ 'TYPE' THEN
136    PUT '$';
137 ELSE
138    put;
139
140 IF eof THEN
141    PUT ";";
142 RUN;
143
```

With the INPUT statement now properly created, we need to find the first row in the expression data ASCII file with usable data. The program that creates the original file uses the keyword “FEATURES” in the first data column to indicate where the data starts. All we need to do is to determine which record number has that and add 1 to get the start of the actual data, store it in a macro variable, and use that as the value of the FIRSTOBS= option when we process the file to create our SAS dataset (❻ and ❼). The two macro variables *cv1* and *cv2* were created in lines 124-128, and they contain the actual variable names that we created for the fields of interest.

```
144 /* Find first DATA line which follows the FEATURES line */
145 DATA _null_;
146 LENGTH firstword $ 20;
147 SET files (FIRSTOBS=&i OBS=&i);
148 INFILE in FILEVAR=fn END=eof PAD MISSOVER LRECL=8192  DLM='09'x;
149 DO UNTIL(eof or firstword EQ 'FEATURES');                    ❻
150    INPUT firstword $;
151    count + 1;
152    IF firstword EQ 'FEATURES' THEN
153       CALL SYMPUT('firstdata',sum(count,1));
154 END;
155 RUN;
156
157 /* read file using INPUT statement created above */
158 DATA firstpass;                                              ❼
159 SET files (FIRSTOBS=&i OBS=&i);
160 INFILE in FILEVAR=fn END=eof PAD MISSOVER LRECL=8192 FIRSTOBS=&firstdata DSD
    DLM='09'x;
161 DO UNTIL(eof);
162    %INCLUDE "f:\bernard\inputline&i..sas";
163    OUTPUT;
164 END;
165 KEEP genename probeUID &cv1 &cv2    ❽;
166 RUN;
```

The final step is to put all these data into one SAS dataset (*gems.maindata*.) It’s a continual series of merges, but the %IF is necessary since you can’t merge one dataset by itself.

```
167 PROC SORT DATA=firstpass;
168 BY genename probeUID;
169 RUN;
170
171 /* If first ASCII file, then create main dataset, otherwise merge into dataset */
172 DATA gems.maindata;
173 %IF &i NE 1 %THEN %DO;
174     MERGE gems.maindata firstpass;
175     BY genename ProbeUID;
176 %END;
177 %ELSE %DO;
178     SET firstpass;
179 %END;
180 RUN;
181
182 %END;
183
184 PROC SORT DATA=gems.maindata;
185 BY genename ProbeUID;
186 RUN;
187
188 %MEND doit;
189 %doit;
```

SUMMARY

This process shows the power of the INPUT statement, and the utility of the macro language to automate a seemingly difficult task. Given the size and inconsistency of the file format itself, the flexibility of the INPUT statement to process part, all, or none of a raw data line, and the capacity to write SAS code from a DATA step and execute it in-line were important to the development of this solution. The solution to this problem was effected with just BASE SAS. Each step of the overall solution solves a specific raw data problem, and these individual methods can be used to solve similar problems by themselves.

While this example is a one-time task, it would be easy enough to incorporate this in a macro structure for reuse, or even as a part of an application for use by people working with the same type of data. With the increase in genomic data availability, and the variability of the file formats produced by genotyping equipment, the development of methods such as this will continue to be necessary.

ACKNOWLEDGEMENTS

This work has been partially funded by: NHLBI grant R25-HL085040.

CONTACT INFORMATION:

Further inquiries are welcome to the authors:

Derek Morgan
E-mail: derek.p.morgan@gmail.com

Bernard Omolo, Ph.D.
Department of Biostatistics
3101 McGavran-Greenberg Hall, CB # 7420
Chapel Hill, NC 27599-7420
E-mail: bomolo@bios.unc.edu