PharmaSUG2011 - Paper AD16

Creating reference amplicons and genotyping using the SAS System

Kevin Viel, Saint Joseph's Translational Research Institute, Atlanta, GA

ABSTRACT

The sequence of DNA nucleotides flanked by the primers is known as an amplicon and is frequently used in alignment-based variation scans. The goal of this paper is to describe two BASE SAS® macros that generate the flanks for each base in the reference amplicon that can be used to make a genotype call and record the result and its associated data in a database. The principle underlying this approach is to indentify loci using the flanks by incorporating them in regular expressions (regexes) used in the PRX functions in BASE SAS. We can genotype single nucleotide substitutions (single nucleotide polymorphisms or SNPs), copy number variants, insertions, deletions, and inversions. By querying the database we can report whether a base is a variant or whether we did not cover it-an essential quality control process.

INTRODUCTION

In research projects and clinical work that use Sanger re-sequencing to perform variation scans of DNA, the sequence of nucleotides flanked by the primers is known as an **amplicon** (**Figure 1**). Thus, the reference sequence of the amplicon is known and is frequently used to anchor an alignment-based variation scan, which usually involves a great degree of manual review by a technician. The manual review is at best slow and at worst susceptible to error, either in the actual calls, the identification of the nucleotide loci, or in the recording of the calls. Often, only the loci with evidence of a variant are recorded.

Figure 1. Reference amplicon F8_A056539L0400 Forward. The first base of the amplicon is nucleotide 56,539 of the reference F8 sequence. The last base of the amplicon is 56,938 for a total length of 400 bp (56,938-56,539+1=400). A separate program determined that the smallest fragment (substring) that matches uniquely within this amplicon allowing at most a one base mismatch is 11 bp in length. Any fragment smaller than this length will match at least twice after allowing for a one base mismatch. In this amplicon, two fragments, AAG_TATAAA and AAAAAGTC_G, of length 10 matched twice within the amplicon.

```
AGAATTTTTCTTCCCAACCTCTCATCTTTTTTTCTCTTATACAGAAGTTATAAAAGTCAATATTTGAACAA
TGGCCCTCAGCGGATTGGTAGGAAGTACAAAAAAGTCCGATTTATGGCATACACAGATGAAACCTTTAAGA
CTCGTGAAGCTATTCAGCATGAATCAGGAATCTTGGGACCTTTACTTTATGGGGAAGTTGGAGACACACTG
TTGGTAAGTTGAAGAAAAGATTTAAGGTCAGGTAAGAAGAAAAAGTCTGGAGAGTTTTGAGTTTCTAAAAT
ACCTCATAATTCAGCCTTGTCTCCAATGGACATGATCTTTTAAAAGCTATAAATGTTACACAAATAATAGC
TGATTGTATGTATTTAAAGTTTGAGTATATAGAATAAAAATTTAA
```

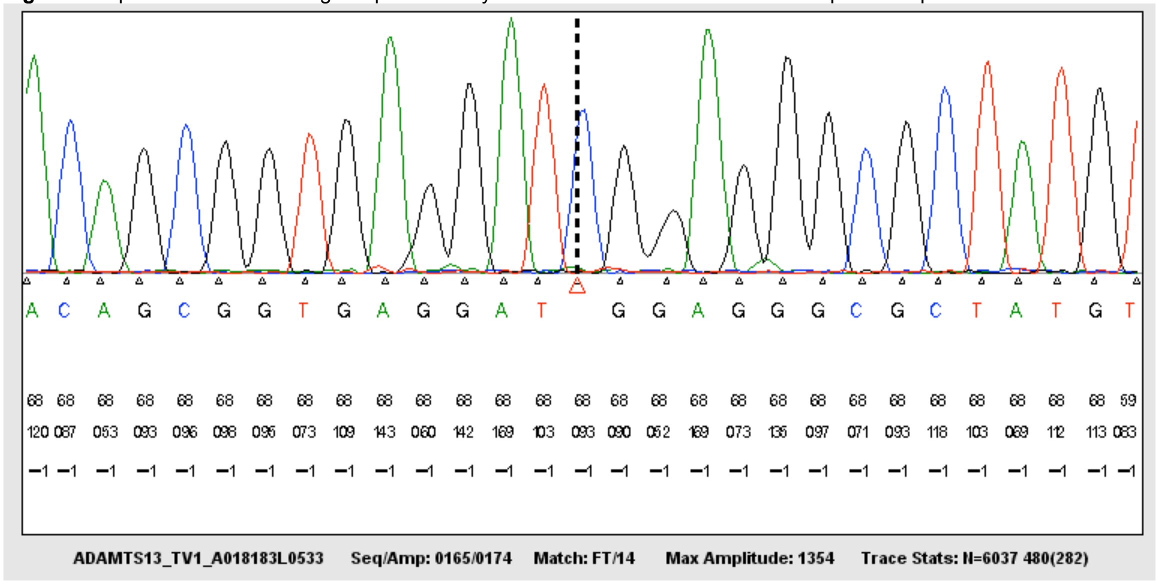
The principle underlying the approach of this paper is to indentify loci using the sequences that flank them, however, since the reference amplicon was created using the reference sequence, either the nucleotide number of the reference sequence or the actual position number in the reference chromosome on which the gene is located are also available. The flanks of a locus create a pattern that is the basis for a **regular expression** (regex) that are then available for use in PRX functions in BASE SAS®[1].

Regexes allow a powerful, concise way to locate the locus of interest in the test sequences. Using them, we can genotype single nucleotide substitutions (single nucleotide polymorphisms or SNPs), copy number variants, insertions, deletions, and inversions. Whenever the regular expression matches in the test sequence, the identity of the nucleotide(s) between the flanks of interests is recorded in the database, whether or not it is a variant. Using this method, each base of a reference amplicon will generate an entry in the database resulting from an objective and define set of criteria even when the regex fails to match, in which case we record data from that event, too.

By simply querying the resulting database, we can report whether a base is a variant and, just as importantly, whether we have sufficient evidence to confirm that it is not. Importantly, this method allows us to state whether a nucleotide in a sequence file matched the reference nucleotide or not or whether we did not cover it, either due to failures at the PCR or sequencing stage or because the nucleotide was in an area of low quality. Using the database we can also quickly generate the chromatogram (**Figure 2**) surrounding the loci without having to open the sequence file and locate it manually.

The **goal** of this paper is to describe two BASE SAS macros that implement the processes described above. The first generates the regexes comprised of the patterns composed of the flanks for each base in the reference amplicons. The second uses regexes to make a genotype call and record the result and its associated data in a database.

Figure 2. A portion of a chromatogram produced by a SAS macro from an .ab1 file and phred output.



DEOXYRIBONUCLEIC ACID (DNA)

DeoxyriboNucleic Acid (**DNA**) is an immensely large linear polymer of four nucleotides or their derivatives covalently linked by **phoshodiester** bonds. The nucleotides comprising DNA are adenine (A), cytosine (C), guanine (G), and thymine (T) and their derivatives, such as methylated cytosine, which plays a role in epigenetics. DNA is a double-stranded molecule formed by hydrogen bonding between A-T and C-G on "complimentary" strands. Knowing the identity of one base reveals the identity of the complimentary base on the opposite strand, thus we frequently talk about **base-pairs** (bp) of DNA. The linear strands of DNA can be over 255,000,000 bp in the case of human Chromosome 01.

If DNA persisted in this linear form, the cells could not contain it, could not adequately control its expression, and the fragile DNA would be destroyed by shearing even within the protection of the cell. DNA is thus coiled around proteins called histones and then further supercoiled. DNA is found in the nucleus and mitochondria of most human cells. With the exception of the **sex-chromosomes,** X and Y, and the mitochondrial DNA, the human **genome** (entire collection of DNA) is diploid-offspring receive one of each **autosomal** chromosome from their mothers and one from their fathers. The mothers also contribute one of their two X chromosomes and their mitochondrial DNA, whereas fathers also contribute either their X or Y chromosomes, thus determining the sex of the resulting embryo. Each human parent contributes 22 autosomes and 1 sex-chromosome for a total of 46 chromosomes in a normal, diploid offspring.

GENES

The major function of DNA is to store the genetic "code" to create and regulate proteins. The loci that encode proteins are known as **genes**. An extreme minority of all DNA is known to encode the amino acids that make up proteins. For example, *F8*, the gene for coagulation Factor VIII that is absent or deficient in the sex-linked bleeding disorder Hemophilia A, is located on the X chromosome and is 186 Kbp (kilo-base-pairs) long. Even so, *F8* is less than 0.1% of the entire X chromosome and only 5% of *F8* actually encodes the amino acids for the resulting FVIII protein. Obviously, stretches of coding DNA (cDNA) are interspersed with stretches of non-coding DNA, **exons** and **introns**, respectively.

In humans, triplets of nucleotides, known as ***codons***, code for one of 20 amino acids. The three bases of a codon may be separated by non-coding DNA in a gene. After the DNA is **transcribed**, cellular processes **splice** the introns from transcription product, bringing together the exons and either adjacent codons or completing a codon interrupted by the intron. In the process of transcription, thymine is replaced with uracil (U); the spliced transcription product is called messenger RNA (**mRNA**), which separate cellular processes then translate into a protein.

Each protein is distinct and thus the genes that encode for them are distinct, too. In fact, gene locations within the genome are well known and further, they correspond to one strand or its reverse compliment. In some cases, similar genes are thought to have arisen by gene duplication, but usually contain distinct variations.

DNA VARIATIONS

Replication of DNA is usually done with high fidelity, but mistakes or damage can occur and not be corrected, and thus be passed to future progeny. We consider five classes of changes that can occur to a linear sequence: single base substitutions, copy number variation, insertion, deletions, and inversions. **Single base substitutions** occur when one nucleotide replaces another, for instance ACT and AGT. **Copy number variants** occur when one base or sequence is repeated a different number of times, for instance CGCG and CGCGCG. **Insertions** occur when a base of sequence appears between what are thought to be two contiguous bases, for instance AT and AAT. **Deletions** are related to insertions, but one or more bases are missing, for instance AAT and AT. Determining whether an insertion or deletion occurs depends on the reference, so these two classes are generally referred to as **INDEL**s or insertion/deletion. Finally, an **inversion** means the sequence between two bases of interests is reversed, for instance ACCTG and ATCCG.

POLYMERASE CHAIN REACTION (PCR) AND SANGER SEQUENCING

As mentioned, genes are not only distinctly located on a chromosome, but also are encoded by one strand. Short stretches of DNA may be amplified in a process called **polymerase chain reaction** (PCR). The process involves locating flanks, or primers, on opposite strands that, together, are unique in the genome. The primers hydrogen bound or anneal to their reverse complimentary sequence in the genomic DNA. The polymerase then replicates the region of DNA adjacent to the primer in the 5' to 3' direction. This also happens in the same reaction vessel with the other primer of the pair, which anneals to the opposite strand. Within a few cycles, the replicated region of DNA dominates in number in an exponential fashion. Within an hour, millions of high fidelity copies of the DNA of interest are created (while the other DNA is more or less discarded).

The exact sequence of this PCR-amplified DNA may be unknown, beyond that of the primers. To determine the base pair sequence of this amplified DNA, another type of cyclical amplification occurs, known as Sanger sequencing. In this case, the nucleotides provided to the reaction vessel also contain modified nucleotides that not only inhibit further polymerization when they are incorporated, but they also are linked to one of four distinct dyes that can be detected by a laser. Using the PCR products as a template, the sequencing reactions generate populations of "fragments" of the exact target sequence that consist of every possible length up to that of the original target sequence, all of which end in a modified nucleotide that can be detected by a laser. In capillaries of specialize instruments, the time needed for the elution of these fragments depends on the size of the fragment (much like high pressure liquid chromatography [HPLC] with capillaries instead of columns). Thus recording the elution time (length) of a fragment and using a laser to identity the modified, last nucleotide, and combining data from all fragments generated, one can decipher the sequence of the target loci. Loci up to 1,200 bp in length can usually be "re-sequenced" in this method. The macros described in this paper use the sequence files, namely the *.ab1[3] files, produced in these types of sequencing runs using Applied Biosystems instruments, such as the 3130xl Genetic Analyzer.

OVERVIEW OF THE MACROS

The purpose of the macro AMPLICONS_FLANK_REGEXES (AFR) is to generate the **regular expressions** (regexes) for each base in the reference amplicon (Figure 1). The pattern for the regex (**Table 1**), consisting of the target base between two flanks, uniquely identifies the target base within the reference genome, by virtue of uniquely identifying it within the reference amplicon, which itself is uniquely located in the reference genome.

Table 1. Three Regexes with capture buffers for the 11th base of F8_A056539L0400 in the forward direction.

Flank_Regex	Amplicon_Position	NN	P	FT	Amplicon_Name	Direction
(AGAATTTTTC)(.)(TCCCAACCTC)	11	56549	154194450	FT/10	F8_A056539L0400	Forward
(.GAATTTTTC)(.)(TCCCAACCTC)	11	56549	154194450	F01/10	F8_A056539L0400	Forward
(AGAATTTTTC)(.)(.CCCAACCTC)	11	56549	154194450	T01/10	F8_A056539L0400	Forward

If a match occurs in the test sequence file, then obtaining the genotype of the base is possible. The size of the flanks was previously determined so that any fragment of that size with at most one mismatched base will match within the reference amplicon only once. This avoids ambiguity; the match criterion employed is stringent because it generally requires two such flanks.

As an example, consider the amplicon: ACGTTGAC. For a flank size of three, the macro would create the following: **.**CGT, A**.**GTT, AC**.**TTG, ACG**.**TGA, CGT**.**GAC, GTT**.**AC, TTG**.**C, and TGA**.**, where the period represents the base for which we would like to obtain a genotype. The macro allows the user to control whether to generate only full-length flanks or allow a partial flank paired with a full-length flank, with the caveat that at least one flank must be full-length and match exactly. The actual regular expression created by the macro for the fourth fragment (the first with full length flanks) is (ACG)(**.**)(TGA). The pairs of parentheses are **capture buffers**.

When the pattern matches successfully, one can find the actual valued matched using the SAS function PRXPOSN(), which is not used in this macro but in BC_PHRED, but knowing that the function will be used illuminates the motivation for the pattern. Presuming a successful match, one knows the identity of the flanks and one might reasonably ask why one should use capture buffers for the flanks, which adds to the required computational resources. This would be true except that the macro also creates flanks in which one mismatch, or "wiggle", is allowed in the pattern at each possible position in the flank for the reason that either noise in the sequence or an actual single base variation (SNP) may exists. An example of a regular expression with a wiggle allowance might be (**.**CG)(**.**)(TGA).

The choice of a period in the regular expression, which matches any one byte character, is a matter of economy. The only characters expected in the sequence are A, C, G, T, and N (however, the sequence in AB1 file uses the IUPAC symbols, for instance S for C or G and W for A or T). Use of the period requires one byte, the same as the base for which it substitutes in the pattern.

The second macro, BC_PHRED (BCP), uses the regexes created by AFR to attempt to genotype and creates a record in the database of that attempt, whether or not it successfully matched. BCP uses two files, phd and poly, created by PHRED[2] (www.phrap.org), a base-calling program. Phd and poly provide the quality scores, the called and uncalled base sequence, and limited trace (chromatogram) data such as the position (elution time) of the bases and the amplitude (relative fluorescence unit) of the peaks at that position. For each test sequence file, the data step in the BCP cycles through the regexes for each base (amplicon position) in the reference amplicon.

The two macro discussed in this paper are part of a suite of SAS programs. The creation of the reference sequence file is the topic of Paper P012 that appears in these proceedings. The successful use of these macros without modification relies on the structure and naming conventions employed in the full suite of programs. The discussion and demonstration of the macros should be comprehensible by only reading this paper, but the author strongly encourages the reader to obtain and study the other paper.

EXPLANATION OF SAS CODE

AMPLICONS_FLANK_REGEX

The MACRO statement that indicates the start of the macro code and names the macro uses **keyword parameters** to give the user flexibility (Lines 1-5). When the user calls the macro, he or she must provide values for the parameters, except for the MIN_FLANK and PARTIAL, which have default values. AMP_DS is the name of the dataset containing the name and direction of the reference amplicons. GENE refers to the reference sequence dataset of the gene of interest (technically, the term "gene" for the purposes of this macro could refer to any reference sequence whether genomic DNA or RNA, if the data structure is correct). The creation of reference sequence is discussed in X in the proceedings of this meeting. MIN_FLANK is the length of the flank to be created. It is called Min for minimum because it refers to the size of the smallest fragment that matches no more than once in the reference amplicon (Figure 1). The last parameter, PARTIAL, allows for the creation of patterns that would allow for an exact match of one flank when the other is absent. Such a match could be useful for the ends of the amplicon, which generally aren't of interest, or to detect INDELs or inversions. By default, this option is "off".

The SQL procedure (lines 8-15) creates two macro variables. Two macro variables that obtain their values by the keyword parameters appear in this SQL procedure. These macro variables are distinguished by the ampersand. Whereas the ampersand is a needed token, the period is optional, but the preferred style of the author. To understand this macro, a brief explanation of the Amplicon_Name is needed. The amplicon name refers to a reference sequence (of a gene), named by the GENE macro keyword parameter (Line 2). The example used in this paper is F8_A056539L0400. The amplicon name reveals the source reference sequence (gene, here *F8*), details about its location in the gene (A056539), and the length of the amplicon (L0400). Since GENE provides the reference sequence data set, only the second part of the name is needed from Amplicon_Name. Its three components are: A or a, the nucleotide number of the starting base, and the Length. Since the subsequent data step applies to all of the amplicons in the data set AMP_DS, the maximum length of the smallest flanks and the amplicons must be determined. "A" signifies that the start of the amplicon occurs after the transcription start site, whereas "a" signifies that the start of the amplicon occurs before the transcription start site (in the promoter or 5' genomic DNA). By the conventions adopted in this approach, the nucleotide number (NN) of bases before the transcription start site are negative-note, that the NN of transcription start site is +1. A NN of zero does not exist by convention requiring special attention in the algorithm and code implementing it.

The first selected column (Line 9) contains both the MAX operator and the MAX() function. The MAX() will chose the maximum value of Smallest_Flank in dataset. If this value is less than MIN_FLANK, then the returned value is MIN_FLANK. The selected second column parses the name, abstracting the length of the amplicon. The length is a zero-padded "number". The use of zero-padding, which can be achieved in SAS with the Zd. FORMAT and the PUT() function. The use of INTO with a colon before the macro variable names is required. The author prefers to indent and align the commas for ease of reading.

Once the maximum sizes of the minimum flank and amplicons have been determined by this SQL procedure, the following data step can create the regexes. The output from this data step is not modest. For each amplicon position, it creates at least 2L + 1 regexes, corresponding to the full-length match patterns for the exact flanks and two each for a wiggle at each position in the flank. Typical genotyping projects might have amplicon lengths ranging from 800 to 1200 base-pairs and require 30-40 amplicons, with both forward and reverse coverage.

The DATA statement (line 17) creates one data set, AMPLICON_ FLANKS, and uses the Data Set Option KEEP to control what variables appear in the output. The variables Amplicon_Name and Direction originate in the AMP_DS input data set. Amplicon_Position is the relative location within the reference amplicon, generated in the data step (Do-Loop starting on Line 97). Flank_Regex is the regex described above, the main product of this data step. NN is the Nucleotide Number relative to the reference sequence specified in GENE. P is the position in the chromosome reference sequence. Importantly, P is the "absolute" identifier; the base to which it refers may be in several different reference gene sequence files with different NN in each, but the value of P will be the same. Finally, FT indicates the type of match and the size of the Smallest_Flank for that amplicon. FT (five prime, three prime) may be used in quality control analyses, for instance by LIMS (Laboratory Information Management Systems) and we may condition on its value to derive the consensus call for that base in a later program of this suite.

The LENGTH statement (lines 21-30) designates the length of multiple variables. Note that the type (numeric or character) and size in bytes must be indicated, but "grouping" is allowed so that only these only have to be specified for the last variable in the group. The two macro variables (Max_Flank and Max_Amp_Size) created in the proceeding SQL statement appear here. Without these macro variables, the required values would not be otherwise known when SAS *compiled* the data step. For length of Flank_Regex, a calculation is required, which is accomplished using the macro function %EVAL, which requires integer values. This length provides for two flanks of size Max_Flank plus a pair of parentheses for enclosing each and a period enclosed by a pair of parentheses. The LENGTH statement specifies three numeric variables (Amplicon_Position, NN, and P). Although the length of numeric variables is rarely an issue if the default length is not used, including them orders the variables in the output dataset and is more a matter of a convenience and preference.

The dataset then creates the hash __AP (Amplicon Position). Hashes are associative arrays that have two major advantages: 1) they can be keyed by any type of variable or combination thereof and 2) their size is dynamic (can change during execution). In contrast, SAS arrays require an integer value for their indexed and the size must be provided at compilation, not at execution. If the key (Amplicon_Position) and values (NN and P) did not appear in the LENGTH statement, their type and length would have been unknown. It is better to explicitly define them if one cannot find it another way, for instance by using a SET statement. The CALL MISSING() call routine sets the values to missing (Line 39) so that SAS does not send a message to the log. The purpose of the hash is to faithfully associate the correct values of NN and P with their amplicon position. This is necessary because 1) we may use both forward and reverse amplicons and 2) the genes may be on the "-" strand of the chromosome in which case as the value of NN increasing the value of P decreases.

The DO-UNTIL loop (lines 42-63) "loads" the regexes into the hash __AP, reading them from the data set named by the macro variable AMP_DS . The loop exits with the value of End is 1 (or any non-zero value, but by virtue of its definition, it can only be 0 or 1). End is created and assigned its value by the END= option to the SET statement (Line 43). When the last observation of the input data set(s) is read, SAS assigns 1 to End. A DO-UNTIL loop checks the condition at the end of the loop. In contrast, a DO-WHILE loop checks the condition at the start of the loop.

Before obtaining the regexes for each position of the amplicon, the data step must first obtain the sequence of the reference amplicon. As mentioned the amplicon name provides pertinent details about the amplicon. Line 44 assigns to S1 the NN of the starting base of the amplicon. Using the argument -1 as the second argument of the first (leftmost) SCAN() function, counts the components separated by the third argument (an underscore) from right to left, i.e. form the end. The result of the first SCAN function (of the form AXXXXXLYYYY) is the first argument of the second SCAN function, which returns the first component when the delimiter is "A", "a", or "L". The value is a string of numbers, potentially left padded with zeroes. The INPUT() function converts this character string to a numeric value (the variable NN is numeric in the GENE data set). If the first position is "a", then starting NN is negative. The values "A" and "a" were chosen instead of “+” and “-“, for instance, because the amplicon names become incorporated into the file names and “A” was chosen for “Amplicon”. Line 46 similarly parses the amplicon name to return the length as a numeric variable (L1). The length of the amplicon includes its first base, so that A0001L09 would include bases 1-9, not 1-10. Since the name of the amplicon was designed to exclude bases covered by the primer, the bases in the reference amplicon are those that are amplified and can vary; one can decipher in this example that the last base of the forward primer was -1 and the first base of the reverse primer was 10, but the lengths of the primers are not known (however, based on the desirable characteristics of the primer chemistry, one has a very good chance of either identifying the primers or suitable substitutions, neither of which are needed for the function of this macro other than in the creation of the names of the amplicon). Thus, at this point the data step has the three pieces of information required to identify the sequence of the amplicon: the starting NN, the length of the amplicon, and the name of the reference sequence to which they pertain.

As mentioned, the NN 0 does not exist by convention. NN -1 is the last base of the promoter and the next base is +1, the transcription start site. Thus for amplicons containing the NN -1 and NN + 1, additional logic would be required to obtain the correct number of bases in the amplicon. To circumvent this, the data step instead employs a counter (__N, Line 47 initializes it to zero). Line 48 initializes Amplicon to missing (character). The last step before the loop to read the GENE data set is to clear the hash (Line 49).

The DO-Loop (also known in SAS-L programming circles as a DOW or Whitlock-DO-Loop) on Lines 51-63 reads the GENE data set, subject to the controls obtain from parsing the amplicon name, and adds the key-value pairs to hash __AP. The GENE data set is indexed by NN. Thus, a SET statement with the KEY= option is used. The DO-Loop initializes NN to S1 (line 51); upon each iteration, the value of NN is increment BY 1 UNTIL the condition that the count (__N) is equal to the value of L1, the length of the amplicon.

The first action of this DO loop is to increment the counter (Line 52). This is done very concisely. This line, in effect, both RETAINs __N and acts as if it employed the SUM() function. Note that SUM(. , 1) returns 1, whereas __N = . + 1 will return missing (.). Since NN is incremented BY 1, it may very well be assigned the value 0. If this occurs, its value is incremented to 1 (Line 54), but no additional increment was made to the counter. The terms “Forward” and “Reverse” become important at this point and merit a short discussion. As mentioned, a gene appears on one strand of DNA or its reverse compliment. Sequencing in both directions is wise because some peaks “resolve” better in one direction than the other. The “Forward” direction is, by the convention of this approach, when the NN increases as one progresses form left to right in the amplicon sequence. Referring to the relative position in the amplicon (Amplicon_Position, lines 55 and 56), another convention adopted keeps the first position of the amplicon as 1. Thus, the first amplicon position of the forward amplicon corresponds to the Lth position of the reverse amplicon. Indeed, the potential for confusion is great, even among experienced users. We therefore maintain the NN and P with the Amplicon_Position, even though this adds to the storage requirement. However, without knowing the source amplicon, one can easily assemble all of the attempted calls for a given base by matching on its NN or P, either across files or projects, respectively. Once the NN and Amplicon_Position is assigned, the reference base and P can be read from the GENE data sets (lines 57-60, with KEY= on Line 59). Note that the code permits a risk in that it does not check the return code (_IORC_) to see if the read was successful. It is incumbent upon the user to be sure that the data sets are in order and that the correct GENE dataset and amplicon names are used-not a trivial issue. As each base is read from the reference sequence, Line 61 concatenates it to the amplicon. The CATS() function concatenates the base to the amplicon after each argument has been stripped (S for STRIP()) of leading and trailing blanks and assigns the returned value to Amplicon, both the variable accepting the assignment and an argument to the function-it is a bit of a SAS “feature” to allow this construct. This line makes the maximum length of the amplicons in the AMP_DS essential; if the length of Amplicon were too short, then SAS would set _ERROR_ to 1 and issue a WARNING to the log. The ADD method (Line 62) adds the hash values NN and P to the hash __AP indexed by the key Amplicon_Position assigning the return code to __RC. Again, no check of the value is made because we assume it is successful.

At the end of the loop, the sequence of Amplicon is known and the hash __AP is “loaded”. In three statements, the data step will be ready to create the regexes for each Amplicon_Position. The first step capitalizes (UPCASE() Line 65) the amplicon sequence. In the GENE dataset, the exonic bases are capitalized whereas all other bases (intronic, promoter, genomic) are lower case. Although, capitalization may appear purely cosmetic since the matching (for instance, the PRXMATCH() function in BCP) is performed insensitive to case, the author purposefully developed this “habit” so that upon potential manual review, the technician is “blinded” to the nature of the base since extra effort or attention might be given to an exonic base. For Reverse amplicons, the reverse compliment of the sequence is obtained (Lines 66-75). The TRANSLATE() function works well and can function on multiple pairs of targets and changes, but the changes are made to lower case. For instance, “T” is changed to “a”. The returned “compliment” value from TRANSLATE() is then converted to capitals and reversed (REVERSE). Finally, leading and trailing blanks are stripped (STRIP). For instance, CAGT would become ACTG. The last statement (Line 76) determines the length of the flank. For some amplicons, especially those that might be entirely within a (large) exon, which typically do not have repetitive sequence, the smallest flank might be less than some desirable number (the author prefers 10). In this case, the macro user has the opportunity to insist upon a stricter minimum criterion. S_F takes the larger of Smallest_Flank (read from AMP_DS) and MIN_FLANK (assigned as a macro keyword parameter), as determined using of the MAX operator. From this point, cycling through each Amplicon_Positon of the Amplicon will create the regexes using three DO-Loops: one for partial five prime flanks (FPF) and full three prime flanks (TPF), one for full FPF and TPF flanks, and one for full FPF and partial TPF. Partial flanks occur when attempting to match bases near the ends of the amplicon. In each case, the pattern is comprised of contiguous sub-sequences that maintain the order.

As the Do-Loop creates each regex is also creates a variable, FT, that indicates the nature of the match and the size of the flanks; each regex includes one exact full flank, but the other may be missing, an exact match of a partial flank, or an exact full length flank with at most one mismatch base. For example, “p5” indicates that the regex is completely missing the FPF and has an exact full length TPF; “P5” indicates that the regex comprises an exact, but less than full length FPF and an exact full length TPF; and F12 indicates that the regex comprises an exact full length FPF and but that the TPF has a mismatch at base 12. The second part of the variable FT indicates length of the flanks. By obtaining a frequency of this variable by base, one may obtain a measure of the strength of support for a true variant and how much noise exists in the test sequences.

The first Do-Loop (Lines 79-94) creates the regexes for the first bases of the amplicon. The loop ends when Amplicon_Position is greater than S_F (smallest flank), so it creates the regexes that have only an absent or less than full length FPF. When the Amplicon_Position is the first base of the amplicon, the FPF is absent and FT is “p5” (line 80), whereas from the second base until the S_Fth base, the FPF is a partial, but exact flank, FT is “P5” (line 81). By using the SUBSTR() function and appropriate starting points and lengths, the sequences of the FPF and TPF are created (lines 83-84 and 85, respectively). Note that the macro has not verified that the length of that amplicon is actually longer than the length of the flanks. Whereas, this would be an easy modification, the author has verified the contents of the data set containing the amplicons that are the subject of the genotyping projects. Finally, the nucleotide number (NN) and absolute chromosomal position (P) are reset to missing (Line 86). Using Amplicon_Position as a key, the corresponding NN and P that are the values for that key in the hash __AP are retrieved using the FIND hash method. Usually, it is very wise to check the return code (hence, __RC), but we are sure that this key-value pair exists in __AP since we assigned it within this data step (Line 62). The regex pattern with capture buffers is concatenated and assigned to the variable Flank_Regex (line 84) and output to the data set (line 85). Regexes are concise and powerful, but may require additional study to use to their potential. The parentheses involved in the capture buffers do not contribute to the length of the match; if one wants to match a parenthesis, then one must use an escape character, for instance “\(“, since a parenthesis is among the regex metacharacters. Another interesting point is the use of a null flank in the case of “p5”, for instance. This is different from matching either a space or a newline. For the purposes of the program, we could forgo its use: ()(.) (2345) and (.)(2345) would work. However, the author retains it for the “symmetry” between the various Do-Loops.

This Do-Loop produces S_F regexes, one for positions 1 through position S_F of the amplicon. As previous stated, the design of amplicons typically allows for the “loss” of the first (and last) 20-30 bases of the amplicon in sequencing. If these bases are of interest, one either redesigns the amplicons or creates second overlapping amplicon. However, it is interesting, if not potentially useful to obtain information on these bases and, if an overlapping amplicon is needed, then these bases are of interest and results from these amplicons might be used.

The next Do-Loop (lines 97-152) creates the patterns for amplicon positions that have two full-length flanks and their “wiggle” variants, i.e. those positions that are within S_F of the ends of the amplicon. Within this loop, we are not worried about attempting to SUBSTR() beyond the end of the variable Amplicon because we set end of the Do-Loop accordingly. For each Amplicon_Position, the assignment statements use the SUBSTR() function to obtain the sequence of the FPF and TPF (Lines 100 and 101, respectively). Note the FPF ends one base before amplicon position and the length of the FPF is S_F: (Amplicon_Position -1) – (Amplicon_Position – S_F) + 1 = S_F – 1 + 1 = S_F, the intended length. The addition of one is not always obvious, but the difference between the position numbers and the number of bases between them differs by one. For example, for a flank that starts at NN = 1 and ends at NN = 10, the number of bases (length) is 10, but the difference between 10 and 1 is 9.

Additional assignment statements that replicate the FPF and TPF as FP_Frag and TP_Frag, respectively, will store the original, exact flank sequences so that the wiggle variants can be created (Lines 103 and 104, respectively). The values of NN and P are again set to missing (Line 105), and the values corresponding to the key Amplicon_Position are retrieved using the FIND hash method. The regex for the pattern with exact, full-length flanks is created and output to the data set (Lines 109-111 and 112, respectively).

At this point, a Do-Loop (lines 115-138) cycles through each position of the flanks. The goal of this Do-Loop is to replace each base of one flank with a period one at a time. Again, in a regex the metacharacter period matches any one-byte value, but we can use it without changes the length of the flank variables. To do so, we use a SUBSTR() function again, but this time, it appears on the right side of the assignment statement (Lines 116-117)-an unusual arrangement for SAS functions. When this function appears on the right side of the assignment statement, it "excises" the string start at the value given as the second argument of the function. The length of the excise string is given as the third. The excised string is replaced with the string that appears on the right side of the equation. During each iteration, the _n_th position of the flank is replaced with a period. To avoid the computation SUBSTRing during each iteration, we instead use a second pair of variables to store the original value, which we reassign after the regex has been built later in the Do-Loop. One has a variety of approaches to accomplish this replacement including CATX(), TRANSLATE(), TRANWRD(), Call Poke(), et cetera. However, the use of SUBSTR(left of =) seems to minimize the number of calculations while clearly indicating its purpose, once, that is, one is accustomed to using it.

It may appear that the substitution of the wiggle in FP_Frag and TP_Frag (Lines 116 and 117, respectively) violates the criterion that allows at most one mismatch in one flank while requiring an exact match in the other. However, FP_Frag pairs with TPF and FPF pairs with TP_Frag (Lines 121-124 and 129-132, respectively), thus each regex pattern contains at most one potential mismatch. Notice that the order of creation an output begins first with the exact match then proceeds to the wiggle. This Do-Loop creates 2 * S_F regexes (one for each position of the flank for each flank). However, in the matching process employed by BCP, once one pattern of the group for the Amplicon_Position matches no further attempts are made.

Lines 138-152 create the regexes for patterns that match only one exact, full-length flank while the other is missing (technically, a null length flank). This is not without reason, even though these calls will not be used to obtain a consensus call. At times, even the best technician cannot help but produce noisy sequencing. Despite the criteria requiring matches on both flanks, the author has witnessed matches to extremely noisy test sequences that are amazing. Although, the author discards calls from noisy flanks, "hits" assure the user that the file indeed was generated from the primers for that amplicon. Further, a real situation occurs in which segments of the reference DNA sequence are absent from the test DNA, an INDEL. When this occurs, no full-length flanks in the area of interest will match, unless by a quirk of fate that the adjacent sequence matches the flank. The regexes generated by this macro are a poor way to genotype INDELS, but by querying the resulting database and noting (repetitive) "gaps", suspicion of an INDEL grows, especially in low noise, high QV regions. Since the macros that generate regexes for both INDELs and inversions are in development, the author has left the choice to produce them as an option in AFR (Line 140, which uses the %IF, the macro version of IF).

Finally, the last Do-Loop (lines 155-171) creates the regexes for the patterns at the end of the amplicon. This is similar to the Do-Loop on Lines 79-94, except the FPF will be full-length. Unlike the sequence at the beginning of the amplicon, the sequences for these bases is usually quite good, unless the amplicon is over 1,000-1,200 base-pairs. Understanding the PCR and sequencing reactions clarifies why this is so. The sequencing reactions create a population of fragments of varying lengths up to the length of the amplicon. The elution times of longer fragments become increasingly indistinguishable from fragments one to a few bases shorter, but out to 1,000 the separation is usually distinct enough that the sequence is clean. Thus, the bases at the end of the amplicon are viable targets, but no full length TPF match is available. Since the sequence can be clean, the question of whether to allow these bases to contribute to the consensus calls rest with whether the user feels that a fragment of length S_F really unique locates the base within the reference amplicon *and* test sequence.

BC_PHRED: The macro consists of a data step followed by two procedures (APPEND and DATASETS). The data step performs three general steps 1) load the regexes created by AFR for a given amplicon and direction into a hash, 2) obtain the phred data for the test sequence, and 3) attempt to obtain a genotype from the test sequence for each position in the reference amplicon. The macro statement (Lines 1-4) names the macro and uses three names keyword parameters: PHRED, the name of the SAS dataset holding the phred data; LN, the name of the library; and OUT_DS, the name of the output dataset. Both PHRED and OUT_DS require the two-level SAS name, if the data are in permanent SAS datasets, but assume the passwords are the same for all data sets. A macro variable stores the password, which is not intended to be a security measure, but rather is a process the author adopts to avoid unintentional changes or deleted data sets.

The DATA statement (Lines 5-9) creates a permanent data set, but it is only intended to be temporary. The KEEP= data set option specifies the variables that will be in the dataset. Only five of these variables are created in this data step: POSITION, QV_FPF, QV_TPF, NOISE_FPF, and NOISE_TPF. The remaining variable listed in the KEEP= option originate in the input data sets.

The LENGTH statement (Lines 12-20) not only creates new variables and specifies their attributes, but it also orders the variables in the output dataset. Of note are __AMPLICON_POLY, _1, _3, __SCAN_FLAG, __DIR, and __AN. __AMPLICON_POLY holds the sequence as originally read from the phred .poly (text) file. It is a character variable with a length of 5000. Since the amplicons of these projects are generally under 1,200 bp in length, even with "noise" at the end of the test sequence files, this length has been more than adequate. _1 and _3 are the actual FPF and TPF held in the capture buffers when the regexes match the test sequences successfully. Again, a length of 60 is more than twice the required length, even for amplicons that cover introns, which typically have more repetitive sequences than exons. __SCAN_FLAG is an indicator variable for a successful match at a given amplicon position whose purpose is to "shut off" further matching attempts. Once a pattern matches, subsequent attempts to match other patterns (those with a wiggle base) are not necessary. __DIR and __AN are also indicator variables that control whether the hash FR should be cleared and reloaded. As long as the incoming test sequences are generated by the same amplicon and direction, the regex patterns loaded into the hash FR do not need to be loaded. When the macro processes a large number of test sequences for a given amplicon and direction, this design should be more efficient than the I/O required to read each regex anew. Both __DIR and __AN are RETAINed (Lines 22); the data step does not reset their values to missing when it (implicitly) loops.

When declaring a hash and defining its keys and values (Lines 28-33), it is best to explicitly state the attributes of the variables that comprise its keys and values. To avoid the "wall paper" of listing its variables in the LENGTH statements and, more importantly, misspecifying them, the use of a SET statement (Line 36) accomplishes this goal since these variables are in data set Amplicons_Flanks. However, the data step does not read a single observation from the data set since the condition _N_ = 0 is never met and thus no execution occurs. During compilation, the attributes of the variables in AMPLICON_
FLANKS become available to the data step by virtue of being "read" in a SET statement-whether the condition is false is not known until execution and even then only we programmers know that it is always false. Several posters on SAS-L point out that coding "IF 0" suffices, but the author adopted the condition _N_ = 0.

The suite of programs involved in this process may use the phred data multiple times so to avoid multiple reads of the raw text file, the phred data were converted to a permanent SAS dataset. The structure of phred data for each test sequence is observations, but to be available to the data step for processing, the data must be held in variables. The data step uses arrays to hold these data (Lines 37-47). The contents of these arrays are reset to missing using a Do-Loop (Lines 49-55) prior to re-loading them with the test sequence-specific data. The inspired programmer might recognize the inefficiency of cycling through &OBS. (5000) for each new test sequence and could use CALL POKE() to accomplish the task. Eventually, the author will have the ambition to be so inspired, but not as of these proceedings.

The data set FILES_AMPLICON identifies the files to be genotyped (Line 58). As the data step reads each observation in this data set, it obtains the amplicon data (name and direction) for each FILE_SEQUENCE_NUMBER (FSN). The phred data set is indexed by the FSN and, thus, only the data pertaining to this FSN are read from it since the KEY= option (Line 63) of the SET statement (Line 62) specified this index. Note that here (Line 66) the author takes the precaution of checking the return code (_IORC_); when obtaining 30,000 files, some of which are 200 Kb in size, especially over networks, occasionally file may be lost or corrupted. This Do-Loop also creates the "linear" test sequences as the concatenation of called bases (Line 67). The SAS System limitation on the length of a character variable greatly limits the very near future use of SAS as a bioinformatics platform for these types of projects; whereas the length DNA in Sanger re-sequencing projects is well under this limit, the so called "next generation sequencing" era may very well generate data well above this limit. The data pertaining to each base, as determined by phred, are loaded into their respective arrays indexed by the position in the test sequence (Lines 68-78). At the end of the loop, when the last observation has been read, the variables are reset for the next iteration (Lines 83-85). Note that this is contingent upon a successful read for that index.

The actual length of the test sequence is assigned as the last (maximum) observation of the phred data (Line 87). Until the observation is read from FILES_AMPLICON, the amplicon and direction to which it pertains is unknown. Once it is read, if they differ (Lines 95-96) from that currently held in the hash FR, then the hash must be cleared and re-loaded (Lines 98-103). They keys for this hash are the Direction, Amplicon_Name, Amplicon_Position, and Count (Lines 29-31). Count is an enumeration of the regexes for each amplicon position, that is the exact match (Count = 1) and those with one mismatch in a flank (Count > 1).

At this point, the data step has the test sequence and the patterns corresponding to each position in the reference amplicon and can attempt to match those patterns within the test sequence. This is accomplished with the Do-Loop in Lines 107-206). The process, however, does not only attempt to match the pattern-it abstracts data about successful matches, attempts subsequent matches in the case of a failure, and records whether a match occurred for a given amplicon position so that further attempts to match other patterns for that amplicon position do not occur.

During each iteration, the variables for these data are reset to missing (Line 108-113). After that step, an inner Do-Loop iterates through the values of COUNT (Lines 117-199). Unlike its outer loop (line 107), the inner loop does not iterate to a set upper limit. Instead, it test the condition provided in parentheses following the UNTIL (Line 117) at the end of each loop immediately prior to looping. The condition is whether the complex key involving COUNT existed in the hash FR. COUNT is incremented by 1 at the end of each loop (Line 107). For a given amplicon position, the number of regexes that correspond to it is unknown, without a calculation. Once the loop cycles through the number of patterns without a match, the next increment of COUNT from the loop will not be a key in FR and the FIND() method will return 0, ending the Do-Loop. Otherwise, a regex pattern will be available and attempts to match in the test sequence can proceed.

The first action of the inner Do-Loop attempts to find the values corresponding to these keys of the hash FR (Line 118). If found (Line 119), an attempt to match its corresponding pattern is made. First, the pattern is compiled to a perl regex (Line 123) using the function RXPARSE() and the starting position of the match is set to 1 (Line 124). Note that the actual match criteria are insensitive to case by virtue of the /i option. The author could have included the both the forward slashes and the /i option in the regexes produced by AFR and it may be worth exploring the relative savings of the two approaches. The choice of starting the search at 1 for each regex, even after a successful match, is simply the preference of the author to be as objective as possible (allowing any match or matches).

The attempt to match is initially accomplished with CALL PRXNEXT() (Line 124-126). __RC_PARSE is the pattern ID of the regex compiled by Line 123. Upon a successful match, CALL PRXNEXT updates __POSITION to the relative position of the match in the string and assigns to START the position of the character next to the match. When this happens, the "hit" data will be abstracted by the Do-While Loop, (Lines 136-193), and __SCAN_FLAG is set to 1 (Line 133) indicating the no more regexes for that amplicon position should be searched by virtue of the LEAVE (Line 198). Importantly, the length of the flank is determined using the variable FT (Line 134).

The number of matches within a test sequence is not guaranteed to be zero or one, despite the criteria on the length of the flanks. Indeed, to the dismay of the author, multiple matches within test sequences have occurred. Rather than being an extremely rare event, such as duplication, such a match most likely represents "bleed through" in which the signal from one capillary is not sufficient shielded from the detector in an adjacent capillary. Also, when the match criteria allow only one ("p5") or a partial flanks ("P3"), multiple hits may also occur. Subsequent attempts to match the regex occur until no further matches occur (POSITION = 0).

Upon a successful match, the condition of the Do-While Loop (Line 127) will be true. In contrast to the Do-Until Loops seen earlier, the condition of a Do-While Loop is tested its commencement. Thus, if __Position = 0, this loop does not execute. The actual sequence of the flanks that the pattern matched are abstracted using PRXPosN() functions (Lines 134 and 135). The POSITION of the base of interest in the test sequence is then calculated (Line 137) as the sum of the __POSITION (starting position of the entire pattern in the test sequence) and the length of _1. Note that if _1 is the null match ("p5"), then the LENGTH() function returns 1 (Length("")=1). To accommodate this, one could use an IF-THEN statement, but a Boolean also suffices. Note that _1 ne " " returns a 1 if _1 is nonmissing, but a 0 if is missing. Once this position is calculated, values for the variables such as the quality values (QV) and area of the uncalled peaks can be assigned from their corresponding arrays (Lines 137-143).

At this point, we have our hands on the flanks and their corresponding data in the arrays. At times, noise masquerades as heterozygous bases. Both the number of bases in the flank that have "high" QV and little "noise" may help future decisions concerning the strength of evidence that the call is true.

Using the length of the flank recorded in the variable __FLANK_LENGTH, and the value of POSITION, the values corresponding to the bases of the flanks can be found in their corresponding arrays. The variables QV_FPF and QV_TPF tally the number of bases in the flanks that have a QV greater than 24, whereas NOISE_FPF and NOISE_TPF tally the number of bases that have uncalled bases with relative peak areas greater than 15%. Two Do-Loops accomplish these calculations for the FPF and TPF (Lines 146-164 and 156-165).

After these data are abstracted, the data are output (Line 166). **Table 2** displays one observation for which a match occurred. At this point, another attempt to match is made (Lines 168-170). If successful, START is again incremented (Line 174) and the values to be abstracted are reset to missing (Line 175-179). In this case __POSITION is greater than zero and the loop has reached its end and returns to Line 127. The data abstraction for this hit, which occurs at the bottom of the loop, actually occurs in the next iteration of the loop.

After exiting from the Do-While Loop, CALL PRXFREE (Line 182) frees the memory allocated to the regex. If a match occurred, then LEAVE (Line 186) exits the loop started on Line 117; no further matches for patterns for that amplicon position will be attempted. If a matched did not, then COUNT is incremented by one and, if the resulting complex key containing it exists in FR, then cycle is repeated. If the loop cycles through all of the regexes for that amplicon position without success, then no hit occurs; FT is

Table 2. An observation resulting from a successful match from BCP.

File_Sequence_Number	NN	Amplicon_Position	Position	
2	61640	94	100	
P	**C_B**	**U_B**	**RA_C**	
154189359	A	N	0.70600	
RA_U	**C_BP**	**U_BP**	**QV**	
-1	1163	-1	57	
QV_FPF	**QV_TPF**	**Noise_FPF**	**Noise_TPF**	**FT**
12	12	0	0	FT/12
A_Amplitude	**C_Amplitude**	**G_Amplitude**	**T_Amplitude**	
24518.03	735.495	438.387	1334.06	

set to missing (Line 191) and the amplicon position is output without any corresponding hit data, but only after a match was attempted for every regex for that position.

The cycle through the AMPLICON_POSITION begun on Line 107 ends at Line 194. At this point, the data step implicitly loops to Line 5, but no action occurs until Line 58 (technically, some of the lines, such as 24 and 25, result in execution, but the checks fail, so no action results). If SAS did not read the last observation in FILES_AMPLICON, then the data step continues, otherwise, it ends at this point. A second macro not covered in this paper, BC_ABI, does not require the phred output but abstracts the data directly from the .ab1 file. To abstract these data, the author uses another macro also not covered in this paper, READ_AB1. Both macros are freely available upon request.

CONCLUSION

The author describes two macros from a suite of programs that generate a database of genotype calls from projects that utilize Sanger re-sequencing and .ab1 files. The first macro (AFR) creates regular expressions (regexes) for each base in a reference amplicon; this macro creates a "family" of regexes for each position based on its contiguous flanks, but each regex requires one full-length flank to match exactly, whereas the second flank may be absent, an exact partial flank, or be full-length with at most one mismatch or wiggle base. This differs from the conventional approach that uses test sequence-wide alignment, such as BLAST[4]. In one respect, the alignment approach may more accurately identify (locate) the base within the reference (anchor) sequence, which is not limited to an amplicon, but may be the whole genome. Within the amplicons of a re-sequencing project this is not likely to represent an advantage. Further, the need for the test sequence to fall within a contig is not required by the approach of this paper, although later processing of the data from a test sequence might exclude it if too many bases result in missing calls. Finally, *every* base of the reference amplicon generates an observation in the resulting database whether the base in the test sequence differs from the reference base or not and whether its patterns match in the test sequence or not. A patient will have at least as many entries for a base as he or she has a test sequence file that covers it.

The author considers the latter to be an essential improvement. In the field of Hemophilia A, for instance, one may be very certain that the reason Factor VIII activity is absent or deficient is due to a DNA defect in the *F8* gene. In variation scans of the approximately 20 Kbp of this 186 Kbp gene that are considered to be classically functional, true variants are often found faithfully, but the presence of false negatives, that is true variants that are not reported, can greatly affect the results of research questions. False negatives most likely result when the bases were not amplified in the test sequence despite being covered, in which case sequence that is not present cannot be aligned and the possible variants that it would contain will not be show up as potential discrepancies in the contig. By creating a database record for each base in the reference amplicon, one can provide a report of the number of times the call was supported and the extent to which coverage may have been insufficient. The author and his groups usually agree that at least two separate, high-quality calls for the same genotype with no unresolved discrepancies are required to generate a consensus call for each position. We relax the phred quality score criterion for potential heterozygous alleles, since phred was originally trained using sequence from clones, which were not diploid. The latest use of these programs by the author resulted in over nine million entries into the database and this was for just one phase involving approximately 50 patients from a project with a target enrollment of 1,200 patients. Clearly, manual review by technicians has limited utility, especially when current technological advances make covering megabase-sized targets feasible, although the read lengths of these instruments pose difficulties for not only this approach but also for the alignment approach, too.

The second macro BCP uses the regexes generated in the first to attempt to obtain a genotype and create an entry for it in a relational database. Most tables in this database have FILE_SEQUENCE_ NUMBER as a key (index). Since an unknown number of files for a given amplicon and direction may be processed, this macro loads the regexes for the latter into a hash and then processes (reads) the incoming files in groups defined by amplicon name and direction, but clears and re-loads the hash when a new group is encountered. If the file data set (Files_Amplicon) is not sorted appropriately, the hash will be cleared and re-loaded many time causing a greatly inefficient process. Clearly two issues affect this approach, on of which is a SAS limitation. The first issue is the number of bases covered in the reference amplicon, which determines whether loading them into a hash is feasible. One alternative would be to read them from a data set; the second would be to generate them on the fly. The second issue is the length of the test sequences. Currently, we are well under the SAS limit on the length of a character variable, which is 32,767 bytes. One alternative is to use perl and an SAS INFILE statement with a PIPE.

BCP reads the list of files, which contains the amplicon name, the direction, and its smallest flank, and then cycles through each position of the reference amplicon to attempt to obtain a genotype. The number of matches of a given regex within the amplicon is not limited, but once on regex of the pattern matches, no further attempts for subsequent regexes for that pattern are attempted. Obviously, if two full-length exact flanks match, every variation of them generated by AFR will also match. Among the full-length matches with a wiggle, if one matches, the others cannot since only one mismatch is allowed in the flanks. The ability to make multiple matches is one way that this approach differs from the conventional alignment approach. True sequence duplication will be missed in the latter, unless multiple alignments are allowed, but that is such an exceedingly rare event as to be almost theoretical. In the author's experience, multiple hits with full-length flanks have represented bleed through. Bleed through can still result in high QV's, so the presence of multiple hits might alert an otherwise unsuspecting user of a potential problem.

The author believes this approach has utility even in the era of next generation sequencing (NGS). First, Sanger re-sequencing (SR) is still a gold standard. It is not uncommon to see SR verification of NGS results. Second, once the target sequence is assembled using NGS, the variants it contains still need to be determined and recorded. It this respect, location is extremely important. Consider the example of *RhD* (CCDS[5] 262.1) and *RhCE* (CCDS 30635.1). Many of us are casually familiar with the Rh factor because of "rejection" in blood transfusing and pregnancy. Rh(-) patients exposed to Rh(+) blood may develop antibodies. Both genes are on Chromosome 01 about 1 Mbp apart and they have an incredibly high level of homology. Many of the SNPs reported in dbSNP[6] for *RhD* actually match the sequence for *RhCE*. **Table 3** provides a few examples that code for non-synonymous-SNPs (ns-SNP) and represent a challenge to the length of the flanks required to uniquely map the bases of interest.

Table 3. Highly homologous DNA sequence of RhD (top) and RhCE (bottom) that code for ns-SNPs

Amino Acid position	DNA sequence
16	`TCCGGCGCTGCCTGCCCCTCTGGGCCCTAACACTGGAAGCA` `TCCGGCGCTGCCTGCCCCTCTGCGCCCTAACACTGGAAGCA`
60	`ATCTGACCGTGATGGCGGCCATTGGCTTGGGCTTCCTCACC` `ATCTGACCGTGATGGCGGCCCTTGGCTTGGGCTTCCTCACC`
68	`GCTTGGGCTTCCTCACCTCGAGTTTCCGGAGACACAGCTGG` `GCTTGGGCTTCCTCACCTCAAATTTCCGGAGACACAGCTGG`
103	`GCTTCCTGAGCCAGTTCCCTTCTGGGAAGGTGGTCATCACA` `GCTTCCTGAGCCAGTTCCCTCCTGGGAAGGTGGTCATCACA`

With appropriate molecular and bioinformatics approaches, the conundrum represented by the extreme example of *RhD* and *RhCE* become manageable, but awareness is important. Nonetheless, this example also illustrates the utility of reference sequence-based database structure for genomic data. The three billion base-pairs of the haploid human genome may very well require a full diploid data base per patient since insertion will represent a general challenge. The situation is compound by potential variants in somatic cells (tumors of cancer offer a vivid illustration) that are not in the germline and the need to store longitudinal results of expression. The later will very likely become a routine diagnostic tool in the near future but it will also generate an immense amount of data.

The sheer volume of data is daunting, but the capabilities are staggering. The approach taken in this suite of program is one that can be extended to the full genome data at the patient level. One may be heartened by the fact that once the sequence of the patient's genome is determined (at birth or *in utero*) the supporting data can be archived. For now, the author keeps these data active and available for review, for which the database and approach described in this paper can be used to easily generate reports and summary statistics for laboratory information management systems (LIMS).

REFERENCES

1. SAS® 9.1.3 Language Reference: Dictionary, Fifth Edition. Copyright © 2002–2006, SAS Institute Inc., Cary, NC, USA. All rights reserved.

2. Ewing B, Green P: Basecalling of automated sequencer traces using phred. II. Error probabilities. Genome Research 8:186-194 (1998).
   Ewing B, Hillier L, Wendl M, Green P: Basecalling of automated sequencer traces using phred. I. Accuracy assessment. Genome Research 8:175-185 (1998).
3. ABIF_File_Format.pdf, ©2009 Life Technologies Corporations. All rights reserved. https://www2.appliedbiosystems.com/support/software_community/ABIF_File_Format.pdf
4. BLAST, http://blast.ncbi.nlm.nih.gov/Blast.cgi
5. Consensus CDS (CCDS), http://www.ncbi.nlm.nih.gov/projects/CCDS/CcdsBrowse.cgi
6. dbSNP, http://www.ncbi.nlm.nih.gov/snp

ACKNOWLEGEMENTS

This work was supported in part by the following grants: NIH-5RC2HL101851-02, NIH-5R01HL072533-04, and NIH-5K08HL071130-05.

These macros and the suite of programs have been an ongoing development for almost a decade. I am indebted to a great number of people who have provided sequencing projects, criticism, and information. Undoubtedly, I may have unintentionally omitted a few people from this list. For general help with the AB1 file, I would like to thank Malay K Basu (malay@bioinformatics.org), the author of ABI.pm (http://search.cpan.org/). I would like to thank Tom Howard, Melinda Epstein, and Richard Hughes of the Greater Los Angeles Veteran Administration Hospital and Shelley Cole, Margie Britten, and Vanessa Ayala of the Texas Biomedical Research Institute (formerly the Southwest Foundation for Biomedical Research) for collaborative efforts on projects and input concerning files and process. Peter Cullinan deserves my gratitude for his technical assistance with this document. Finally, I am indebted to the countless members of both the SAS-L (http://www.listserv.uga.edu/archives/sas-l.html) and ABRF (www.abrf.org) listservs.

RECOMMENDED READING

The approach described in this paper has been employed in two projects that have generated publications. A third project is ongoing. Reading these papers is not necessary to understand and critique the approach and code, but it may give the reader a feel for the potential applications of this suite.

Viel, K.R., et al., A sequence variation scan of the coagulation factor VIII (FVIII) structural gene and associations with plasma FVIII activity levels. Blood, 2007. 109(9): p. 3713-3724.

Viel, K.R., et al., Inhibitors of factor VIII in black patients with hemophilia. N Engl J Med, 2009. 360(16): p. 1618-27.

Contact Information

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

Name:	Kevin Viel
Enterprise:	Saint Joseph's Translational Research Institute
Address:	5671 Peach-Dunwoody Road, NE Suite 330
City, State ZIP:	Atlanta, GA 30342
Work Phone:	678-843-6076
Fax:	678-843-6153
E-mail:	kviel@sjha.org, genepistat@gmail.com
Web:	www.sjtri.org

```
%Macro Amplicons_Flank_Regexes ( Amp_DS    =
                                 , Gene      =
                                 , Min_Flank = 10
                                 , Partial   =  0
                                 ) ;

  /* Example: Amplicon_Name = F8_A056539L0400 */
  Proc SQL NoPrint ;
    Select ( Max( Smallest_Flank ) Max &Min_Flank. )
         , Max( Scan( Scan( Amplicon_Name , -1 , "_" ) , 2 , "AaL" ))
      Into : Max_Flank
         , : Max_Amp_Size
    From &Amp_DS.
    ;
  Quit ;

  Data Amplicons_Flanks ( Keep = Amplicon_Name Direction Amplicon_Position
                                 Flank_Regex NN P FT
                        ) ;

    Length FP_Frag
           TP_Frag           $ &Max_Flank.
           /* Example: (123)(.)(567)*/
           Flank_Regex       $ %Eval( 2 * ( &Max_Flank. + 2 ) + 3 )
           Amplicon          $ &Max_Amp_Size.
           Amplicon_Position
           NN
           P                 8
           FT                $ 6
           ;

    If _n_ = 1
    Then
      Do ;
         Declare Hash __AP() ;
         __RC = __AP.DefineKey ( "Amplicon_Position" ) ;
         __RC = __AP.DefineData( "NN" , "P" ) ;
         __RC = __AP.DefineDone() ;
         Call Missing( Amplicon_Position , NN , P ) ;
      End ;

      Do Until ( End ) ;
        Set &Amp_DS. End = End ;
        S1 = Input( Scan( Scan( Amplicon_Name , -1 , "_" ) , 1 , "AaL" ) , 8. ) ;
        If Scan( Amplicon_Name , -1 , "_" ) =: "a" Then S1 = S1 * -1 ;
        L1 = Input( Scan( Scan( Amplicon_Name , -1 , "_" ) , 2 , "AaL" ) , 8. ) ;
        __N = 0 ;
        Amplicon = "" ;
        __RC = __AP.Clear() ;

        Do NN = S1 By 1 Until ( __N => L1 ) ;
          __N + 1 ;
          /* In Case the amplicon tranverses +1 */
          If NN = 0 Then NN = 1 ;
          If Direction = "Forward" Then Amplicon_Position = __N ;
           Else If Direction = "Reverse" Then Amplicon_Position = L1 - __N + 1 ;
          Set Seq.&Gene. ( PW    = &PW.
                           Keep  = Base NN P
                         ) Key   = NN
                         ;
          Amplicon = CatS( Amplicon , Upcase( Base )) ;
          __RC = __AP.Add() ;
        End ;

        Amplicon = Upcase( Amplicon ) ;
        If Direction = "Reverse"
        Then Amplicon = Strip( Reverse( Upcase( Translate( Upcase( Amplicon )
                                                         , "a" , "T"
                                                         , "c" , "G"
                                                         , "g" , "C"
                                                         , "t" , "A"
                                                         )
                                              )
                                      )
                             ) ;
        S_F = Smallest_Flank Max &Min_Flank. ;

        /* Partial FPF */
        Do Amplicon_Position = 1 to S_F ;
          If Amplicon_Position = 1 Then FT = CatX( "/" , "p5" , Put( S_F , z2. )) ;
           Else FT = CatX( "/" , "P5" , Put( S_F , z2. )) ;
          If Amplicon_Position - 1 > 0
          Then FPF = Substr( Amplicon , 1 , Amplicon_Position - 1 ) ;
           Else FPF = "" ;
          TPF = Substr( Amplicon , Amplicon_Position + 1   , S_F ) ;
          Call Missing( NN , P ) ;
          __RC = __AP.Find() ;
          /* Exact flanks */
          Flank_Regex = CatX( "(.)"
                            , CatS( "(" , FPF , ")" )
                            , CatS( "(" , TPF , ")" )
                            ) ;
          Output ;
      End ; /* Cycled through Amplicon_Position for Partial FPF */

      /* Full Match */
      Do Amplicon_Position = S_F + 1 to Length( Amplicon ) - S_F ;
        FT = CatX( "/" , "FT" , Put( S_F , z2. )) ;
        /* Example: S_F = 10, FPF = 1-10, Amplicon_Position = 11, TPF = 12-21 */
        FPF = Substr( Amplicon , Amplicon_Position - S_F , S_F ) ;
        TPF = Substr( Amplicon , Amplicon_Position + 1   , S_F ) ;
        /* Flank_Regex */
        FP_Frag = FPF ;
        TP_Frag = TPF ;
        Call Missing( NN , P ) ;
        __RC = __AP.Find() ;

        /* Exact flanks */
        Flank_Regex = CatX( "(.)"
                          , CatS( "(" , FPF , ")" )
                          , CatS( "(" , TPF , ")" )
                          ) ;
        Output ;

        Do _n_ = 1 To S_F ;
          SubStr( FP_Frag , _n_ , 1 ) = "." ;
          SubStr( TP_Frag , _n_ , 1 ) = "." ;

          /* Wiggle in FPF */
          FT = CatX( "/" , CatS( "F" , Put( _n_ , z2. )) , Put( S_F , z2. )) ;
          Flank_Regex = CatX( "(.)"
                            , CatS( "(" , FP_Frag , ")" )
                            , CatS( "(" , TPF , ")" )
                            ) ;
          Output ;

          /* Wiggle in TPF */
          FT = CatX( "/" , CatS( "T" , Put( _n_ , z2. )) , Put( S_F , z2. )) ;
          Flank_Regex = CatX( "(.)"
                            , CatS( "(" , FPF , ")" )
                            , CatS( "(" , TP_Frag , ")" )
                            ) ;
          Output ;

          FP_Frag = FPF ;
          TP_Frag = TPF ;

        End ; /* Cycled through size of S_F */

        %If &Partial ne 0
        %Then
           %Do ;
               /* FPF Only */
               Flank_Regex = CatS( "(" , FPF , ")(.)()" ) ;
               FT = CatX( "/" , "F0" , Put( S_F , z2. )) ;
               Output ;
               /* TPF Only */
               Flank_Regex = CatS( "()(.)(" , TPF , ")" ) ;
               FT = CatX( "/" , "0T" , Put( S_F , z2. )) ;
               Output ;
           %End ;
        End ; /* Cycled through Amplicon_Position */

        /* Partial TPF */
        Do Amplicon_Position = Length( Amplicon ) - S_F + 1 To Length( Amplicon ) ;
          If Amplicon_Position < Length( Amplicon )
          Then FT = CatX( "/" , "P3" , Put( S_F , z2. )) ;
           Else FT = CatX( "/" , "p3" , Put( S_F , z2. )) ;
          FPF = Substr( Amplicon , Amplicon_Position - S_F , S_F ) ;
          If Amplicon_Position + 1 <= Length( Amplicon )
          Then TPF = Substr( Amplicon , Amplicon_Position + 1   ) ;
           Else TPF = "" ;
          Call Missing( NN , P ) ;
          __RC = __AP.Find() ;
          /* Exact */
          Flank_Regex = CatX( "(.)"
                            , CatS( "(" , FPF , ")" )
                            , CatS( "(" , TPF , ")" )
                            ) ;
          Output ;
        End ; /* Cycled through Amplicon_Position For Partial TPF */
      End ; /* Cycled through Amplicons */
  Run ;
%MEnd Amplicons_Flank_Regexes ;
```

```
%Macro BC_Phred ( Phred      = /* PATH._03_F8_Phred */
                , LN         =
                , Out_DS     = /* PATH._04_F8_BC_Base */
                ) ;
   Data &LN..BC_Base_Tmp ( Keep = File_Sequence_Number NN Amplicon_Position Position
                                  P C_B U_B RA_C RA_U C_BP U_BP QV FT QV_FPF QV_TPF
                                  Noise_FPF Noise_TPF
                                  A_Amplitude C_Amplitude G_Amplitude T_Amplitude
                         )
        ;

    Length File_Sequence_Number NN Amplicon_Position Position P      8
           C_B U_B                                                   $    1
           RA_C RA_U C_BP U_BP QV QV_FPF QV_TPF Noise_FPF  Noise_TPF 8
           __Amplicon_Poly                                           $ 5000
           _1 _3                                                     $   60
           __Scan_Flag                                               $    1
           __Dir                                                     $    7
           __AN                                                      $  100
           ;

    Retain __Dir __AN ;

    If _n_ = 0 Then Set Amplicons_Flanks ;
    If _n_ = 1
    Then
      Do ;
      Dcl Hash FR ( Ordered : "Yes" ) ;
      __RC_Hash = FR.DefineKey ( "Direction" ,  "Amplicon_Name"
                               , "Amplicon_Position" , "Count"
                               ) ;
      __RC_Hash = FR.DefineData( "Flank_Regex" , "FT" , "P" , "NN" ) ;
      __RC_Hash = FR.DefineDone() ;
   End ;

    /* Poly */
    Array CB( &Obs. )   $  1 ;                      /* Called base */
    Array UB( &Obs. )   $  1 ;                      /* Uncalled base */
    Array CBRA( &Obs. ) ;                           /* Called Base Relative Area */
    Array UBRA( &Obs. ) ;                           /* Uncalled Base Relative Area */
    Array CBP( &Obs. )  ;                           /* Called Base Pos */
    Array UBP( &Obs. )  ;                           /* Uncalled Base Relative Area */
    Array A_Amp ( &Obs. ) ; Array C_Amp ( &Obs. ) ; /* Amplitude */
    Array G_Amp ( &Obs. ) ; Array T_Amp ( &Obs. ) ; /* Amplitude */

    /* PHD */
    Array Qual( &Obs. ) ; /* Quality Values */

    Do _n_ = 1 To &Obs. ;
      CB( _n_ )     = "" ; UB( _n_ )     = "" ;
      CBRA( _n_ )   = .  ; UBRA( _n_ )   = .  ;
      CBP( _n_ )    = .  ; UBP( _n_ )    = .  ;
      A_Amp ( _n_ ) = .  ; C_Amp ( _n_ ) = .  ; G_Amp ( _n_ ) = .  ; T_Amp ( _n_ ) = .  ;
      Qual( _n_ )   = .  ;
    End ;

    /* Source of FSN, Amplicon_Name, Direction, Path, and File */
    Set Files_Amplicon ;

    /* Obtain the Sequence and Base data */
    Do Until ( End ) ;
      Set &Phred. ( PW = &PW. )
          Key = File_Sequence_Number
          End = End
          ;
      If __OK ne "1" And _IORC_ = %SysRC(_SOK) Then __OK = "1" ;
      __Amplicon_Poly = CatS( __Amplicon_Poly , Called_Base ) ;
      CB   ( Position ) = Called_base                 ;
      UB   ( Position ) = Uncalled_base               ;
      CBRA ( Position ) = Called_base_rel_peak_area   ;
      UBRA ( Position ) = Uncalled_base_rel_peak_area ;
      CBP  ( Position ) = Called_base_pos             ;
      UBP  ( Position ) = Uncalled_base_pos           ;
      Qual ( Position ) = QV                          ;
      A_Amp( Position ) = A_Amplitude                 ;
      C_Amp( Position ) = C_Amplitude                 ;
      G_Amp( Position ) = G_Amplitude                 ;
      T_Amp( Position ) = T_Amplitude                 ;
    End ;
    If End And __OK = "1"
    Then
      Do ;
         __OK    = "" ;
         _Error_ = 0 ;
         End     = 0 ;
      End ;
    Stop = Position ;

    /* Obtain the Flank Regex's */
    If      __Dir ne Direction
       Or   __AN  ne Amplicon_Name
    Then
      Do ;
        __RC_Clear = FR.Clear() ;
        __Dir      = Direction     ;
        __AN       = Amplicon_Name ;
        /* Load Regex */
        Do Until ( End_AF ) ;
          Set Amplicons_Flanks Key = DAN End = END_AF ;
          /* If the key is found in Amplicons_Flanks */
          If _IORC_ = %SysRC(_SOK) Then __RC_Add = FR.Add() ;
          Else _ERROR_ = 0 ;
        End ;
   End ;
         /* Attempt to match the Flank_Regex within __Amplicon_Poly */
    Do Amplicon_Position = 1 To Input(Scan(Amplicon_Name,2,"L") , 8. ) ;
      Call Missing ( C_B , U_B , RA_C , RA_U , C_BP , U_BP , QV
                   , A_Amplitude , C_Amplitude , G_Amplitude
                   , T_Amplitude
                   , QV_FPF , QV_TPF , Noise_FPF , Noise_TPF
                   , Position , __Scan_Flag
                   ) ;
       /* Cycle through the regexes for each Amplicon_Position until one matches.  */
       /* Amplicon_Position until one matches.  When one matches,                  */
       /* __Scan_Flag = "1" then we LEAVE the Count loop                           */
      Do Count = 1 By 1 Until ( __RC_Count_Find ne 0 ) ;
        __RC_Count_Find = FR.Find() ;
        If __RC_Count_Find = 0
        Then
          Do ;
             Start = 1 ;
             __RC_Parse = PRXParse( CatS( "/" , Flank_Regex , "/i" )) ;
             Call PRXNext( __RC_Parse , Start , Stop , __Amplicon_Poly
                         , __Position , Length
                         ) ;
             Do While ( __Position > 0 ) ;
               /* Start the search for a subsequent match */
               /* immediately after the match             */
               Start          = __Position + 1 ;
               __Scan_Flag    = "1" ;
               __Flank_Length = Input( Scan( FT , 2 , "/" ) , 8. ) ;
               /* Locus details */
               _1 = PRXPosN( __RC_Parse , 1 , __Amplicon_Poly ) ;
               _3 = PRXPosN( __RC_Parse , 3 , __Amplicon_Poly ) ;
               /* Position of the base of interest */
               Position = __Position + Length( _1 )*( _1 ne " " ) ;
               C_B         = CB(    Position ) ; U_B         = UB(    Position ) ;
               RA_C        = CBRA(  Position ) ; RA_U        = UBRA(  Position ) ;
               C_BP        = CBP(   Position ) ; U_BP        = UBP(   Position ) ;
               QV          = Qual(  Position ) ;
               A_Amplitude = A_Amp( Position ) ; C_Amplitude = C_Amp( Position ) ;
               G_Amplitude = G_Amp( Position ) ; T_Amplitude = T_Amp( Position ) ;
               /* FPF QV Noise */
               QV_FPF    = 0 ;
               Noise_FPF = 0 ;
               Do _n_ = Position - __Flank_Length To Position - 1 ;
                 If 0 < _n_ <= Stop
                 Then
                   Do ;
                      QV_FPF    + ( Qual( _n_ ) > 24   ) ;
                      Noise_FPF + ( UBRA( _n_ ) > 0.15 ) ;
                   End ;
               End ;
               /* TPF QV Noise */
               QV_TPF    = 0 ;
               Noise_TPF = 0 ;
               Do _n_ = Position + 1 To Position + __Flank_Length ;
                 If 0 < _n_ <= Stop
                 Then
                   Do ;
                      QV_TPF + ( Qual( _n_ ) > 24 ) ;
                      Noise_TPF + ( UBRA( _n_ ) > 0.15 ) ;
                   End ;
               End ;
               Output ;
               /* Search for next match */
               Call PRXNext( __RC_Parse , Start , Stop , __Amplicon_Poly
                           , __Position , Length
                           ) ;
               If __Position > 0
               Then
                 Do ;
                    Start = __Position + 1 ;
                    Call Missing ( C_B , U_B , RA_C , RA_U , C_BP , U_BP , QV
                                 , A_Amplitude , C_Amplitude , G_Amplitude , T_Amplitude
                                 , QV_FPF , QV_TPF , Noise_FPF
                                 , Noise_TPF , Position
                                 ) ;
                End ; /* __Position > 0 */
             End ; /* WHILE LOOP: __Position > 0 */
             Call PRXFree( __RC_Parse ) ;
          End ; /* __RC_Count_Find = 0 */
          /* If one of the regex's match, then do not attempt */
          /* to match the others for that position            */
          If __Scan_Flag = "1" Then Leave ;
      End ; /* Cycled through Count */
      If __Scan_Flag = ""
      Then
        Do ;
           FT = " " ;
           Output ;
        End ;
    End ; /* Cycled through Amplicon_Position */
  Run ;

  /****************/
  Proc Append Base = &Out_DS. ( PW = &PW. )
              Data = &LN..BC_Base_TMP
             ;
  Run ;

  Proc Datasets Library = &LN. NoList ;
    Delete BC_Base_TMP ;
  Quit ;
%MEnd BC_Phred ;
```