A Theoretical Understanding of 2 Base Color Codes and Its Application to Annotation, Error Detection, and Error Correction

Methods for Annotating 2 Base Color Encoded Reads in the SOLiD™ System

Heinz Breu

Introduction

The SOLiD™ System enables massively parallel sequencing of clonally amplified DNA fragments linked to beads. This unique sequencing methodology is based on sequential ligation of dye-labeled oligonucleotide probes whereby each probe assays two base positions at a time. The system uses four fluorescent dyes to encode for the sixteen possible two-base combinations. This unique approach employs a scheme that represents a fragment of DNA as an initial base followed by a sequence of overlapping dimers (adjacent pairs of bases). The system encodes each dimer with one of four colors using a degenerate coding scheme that satisfies a number of rules. A single color in the read can represent any of four dimers, but the overlapping properties of the dimers and the nature of the color code allow for error-correcting properties. In this document, we discuss the theory that explains these error-correcting properties, show how to correct the misapplications of these properties, and describe software algorithms to utilize and verify the 2 base encoding scheme. For example, we can identify and annotate isolated erroneous color calls, as well as color-reads that correspond to isolated blocks of adjacent nucleotide variants from a reference, most realistically one, two, or three, but as many as the application might require.

Constructing the 2 Base Color Code

The SOLiD System's 2 base color coding scheme is shown in Figure 1.

code	0	1	2	3
dye	FAM	Cy3	TXR	Cy5
	AA	AC	AG	AT
	CC	CA	GA	TA
	GG	GT	CT	CG
	TT	TG	TC	GC

Figure 1: SOLiD™ System's 2 base Coding Scheme. The column under code *i* lists the corresponding dye and the di-bases (adjacent nucleotides) encoded by color *i*. For example, GT is labeled with Cy3 and coded as "1".

Use the following steps to encode a DNA sequence ATCAAGCCTC*:

1. start at the 5' end,
2. replace the di-base AT at this position with its corresponding code 3 from the table,
3. advance by one base, which exposes the TC di-base, and
4. continue, as shown below.

Base Sequence: A T C A A G C C T C
Color String: 3 2 1 0 2 3 0 2 2

This process encodes a *k*-mer of bases as a (*k*-1)-mer of colors. Although this color string codes for four different *k*-mers, knowledge of the type and position of any of its *k* bases helps to encode the sequence. For SOLiD sequencing applications, prepend the leading base to result in a *k*-mer A321023022 (from the example above) from which the base sequence can be reconstructed.

*The example depicted here uses an A as the first base. In practice, the current chemistry on the SOLiD™ System uses a T as the first base.

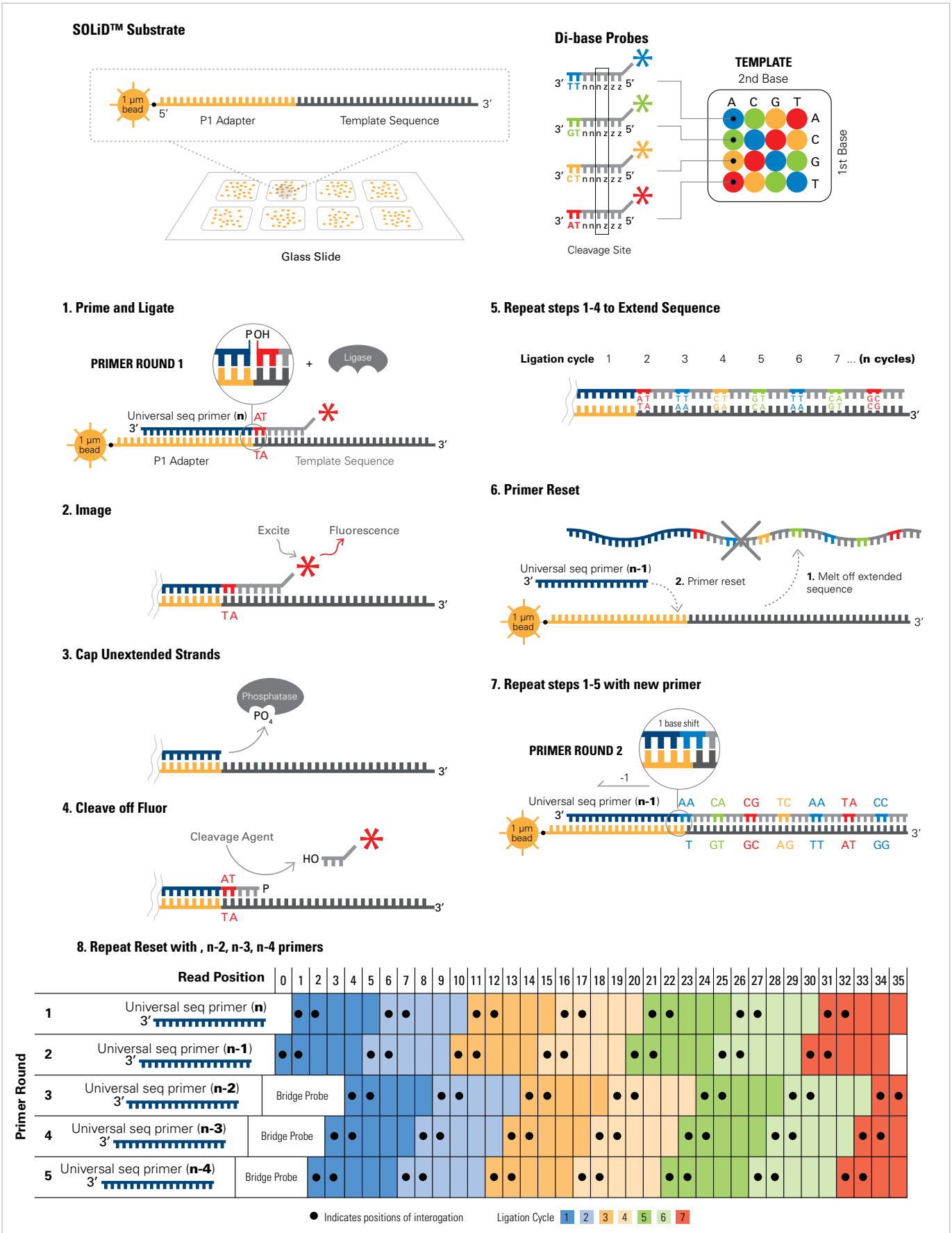

Figure 2. Ligation based sequencing with di-base probes using the SOLiD System. This schematic shows bases interrogated by the di-base probes at positions 1 and 2.

The SOLiD™ System generates its reads in precisely this encoded form. One way of accomplishing this is shown in Figure 2.

Requirements/Properties for a 2 Base Color Code Scheme

The 2 base color coding scheme possesses certain properties that will be discussed later in the document. Interestingly, this scheme is essentially the only code that satisfies these properties. This can be observed by treating the properties as requirements and constructing the color code from them. This document deals *only* with bases, not other IUB (International Union of Biochemistry) codes. So let B = {A, C, G, T}. The color code should satisfy the following requirements:

For all bases *b, d, e* in B:

1. The available colors are 0, 1, 2, and 3:
   color (*bd*) ∈ {0, 1, 2, 3}.
2. Two different di-bases that have the same first base get *different* colors: color (*bd*) ≠ color (*be*) if $d \neq e$.
   For example, color (AC) ≠ color (AG).
3. A di-base and its reverse get the *same* color:
   color (*bd*) = color (*db*).
   For example, color (AC) = color (CA).
4. Monodibases get the *same* color:
   color (*bb*) = color (*dd*).

That is, color (AA) = color (CC) = color (GG) = color (TT). The following are not requirements, but interesting properties that follow from these four. Property 5 follows from requirements 2 and 3, and will make our construction easier.

5. Two different di-bases that nevertheless have the same second base get *different* colors:
   color (*bd*) ≠ color (*ed*), if $b \neq e$.

For example, color (AC) ≠ color (TC). Property 6 also follows from requirements 1-4, but it is most easily verified against the completed code (Figure 3, Panel E).

6. A di-base and its complement get the *same* color:
   color ($b^c d^c$) = color ($d^c b^c$).
   For example, color (AC) = color (TG).

Satisfying the Requirements for a 2 Base Coding System

The remainder of this document uses a notation different from Figure 1. Figure 3 lists the *colors* for each di-base. For example, the value in row C and column T will be the color (2) for di-base CT (Figure 3, Panel A). Requirements 1 and 2 require that all colors are present in the first row. Because the system can use any one-to-one mapping between the actual dyes and the labels 0, 1, 2, and 3 (provided that requirements 1 and 2 remain satisfied) then the first row (row A) can be

		Second Base			
		A	C	G	T
First Base	A				
	C				2
	G				
	T				

Panel A

		Second Base			
		A	C	G	T
First Base	A	0	1	2	3
	C				
	G				
	T				

Panel B

		Second Base			
		A	C	G	T
First Base	A	0	1	2	3
	C	1			
	G	2			
	T	3			

Panel C

		Second Base			
		A	C	G	T
First Base	A	**0**	**1**	**2**	**3**
	C	**1**	**0**		
	G	**2**		**0**	
	T	**3**			**0**

Panel D

		Second Base			
		A	C	G	T
First Base	A	0	1	2	3
	C	1	0	3	2
	G	2	3	0	1
	T	3	2	1	0

Panel E

Figure 3. Requirements that Assign the Color for a 2 Base Code.

labeled as shown (Figure 3, Panel B.) Requirement 3, that color (*bd*) = color (*db*), gives a unique labeling for column A (Figure 3, Panel C). Requirement 4, that color (*bb*) = color (AA), gives a unique labeling for the diagonal (Figure 3, Panel D). Finally, requirements 1, 2, and 5, state that every color must appear in every row and every column exactly once (Figure 3, Panel E). The table (Figure 3, Panel E) is easy to memorize and work with because, by virtue of Property 3, one can think of di-bases as two-element sets for assigning colors. The di-bases starting with 'A' get colors 0, 1, 2, and 3 respectively. Therefore:

0. AA, CC, GG, TT all get color 0,

1. AC and CA get color 1, and so must GT and TG,

2. AG and GA get color 2, and so must CT and TC,

3. AT and TA get color 3, and so must CG and GC.

Determining the Color String of the Sequence

Without the leading base, it is not possible to determine if a particular DNA sequence is GC-rich from its color string alone. Here, we will show that for any GC-rich sequence S, there exists another sequence S', with exactly the same color string. Make the new sequence S' from S by replacing A with G, C with T, G with A, and T with C (i.e., replace the base with the other purine or pyrimidine). The result is an AT-rich sequence that has the same color string.

Example: S = G C G C A G G G T C C T C C T
3 3 3 1 2 0 0 1 2 0 2 2 0 2
S'= A T A T G A A A C T T C T T C

This is because, if a di-base *bd* has a color *k*, then so does its replacement *b'd'*. For example: 0: GG → AA, 3: GC → AT, 2: AG → CT, 1: AC → GT. If the di-base in S has color 0, then its bases are equal and remain so in S'. If it has color 3, its bases are complementary and remain so in S'. If it has color 2 and its bases are both purines, they will both be pyrimidines in S', and vice versa. Finally, if it has color 1 then it is AC, CA, GT, or TG, and so replaced by GT, TG, AC, or CA respectively, all of which again have color 1.

Colors as Transformations of Bases

Up until this point, colors are assigned as a result of an encoding process, either a specific chemical one, like the SOLiD sequencing process, or a purely mathematical one. Here, there is just one function, color: B × B → {0, 1, 2, 3}, that maps di-bases to colors, e.g., color (CG) = 3.

For transformation of bases, color can be represented in a different way. Each color *d* is a function f_d: B → B that "transforms" base *u* to base *v*. There can be any number of colors, but in this case there are four. The transformations are also specified in Figure 3, Panel E. To transform base *u* with color *d*, look up color *d* in row *u* and report the unique column (by Requirement 2) in which it resides. For example, color 3 transforms base C to base G, which can be written in different ways: "f_3(C) = G", "C3 = G", "C3G", or even "C_3G".

Swapping Rules

The following rules can be applied to verify the observations from Figure 3:

1. Color 0 is the identity function.
   That is $f_0(b) = b$ for every base *b*.

2. Color 1 swaps A with C, and it swaps G with T.
   For example, f_1(A) = C.

3. Color 2 swaps A with G, and C with T.

4. Color 3 swaps A with T, and C with G.

Function Composition on Colors

Strings of colors can also be treated as transformations by simply applying one color transformation after another. This is how to decode a color read. For example, to *decode* A321023022 apply color 3 to A to get f_3(A) = T. Then apply the next color, 2, to this result, to get f_2(T) = C. Continuing in this way, decode all bases, including the last:

A T C A A G C C T C
3 2 1 0 2 3 0 2 2

That is, A321023022 decodes to ATCAAGCCTC.

To compose all color functions in an entire string, the next step is to ignore intermediate bases. Just as a single color transforms one base into another, so does a string of colors. The example above transforms the first base 'A' into the last base 'C' of the sequence. This is true of the whole string and also of substrings. In the example above, we can think of the substring 102 as transforming C into G:

A A **C** A A **G** C C T C
0 1 **1 0 2** 3 0 2 2

C1 → A0 → A2 → G

⊕	i	v	h	r
i	i	v	h	r
v	v	i	r	h
h	h	r	i	v
r	r	h	v	i

Panel A

⊕	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

Panel B

Figure 4. Addition Table to Obtain the Code for Strings of Colors as Transformations of Bases. (A) The original Klein four-group addition table**.** This addition table has been obtained by the Klein four-group **http://en.wikipedia.org/wiki/Klein_four-group**, which is the symmetry group of a rectangle. (B) The corresponding addition table for strings of colors.

For input C color string **102** acts just like color 3, and this also transforms C to G. Also, **102** acts just like color 3, for all inputs A, C, G, and T.

To understand this concept, begin by composing just two adjacent colors as follows:

$C10 = f_0 \circ f_1(C) = f_0(f_1(C)) = f_0(A) = A = f_1(C)$

In this example, color string 10 behaves just like the single color 1, which also maps C to A. Using the swapping rules, color string 10 behaves like color 1 for all input bases. Color 1 swaps A with C, and G with T. Color 0 does not swap any of the bases. The rules for all pairwise combinations of colors, showing that each two-color string behaves like a particular single color, have already been determined in the **Klein four-group**. The Klein four-group (**http://en.wikipedia.org/wiki/Klein_four-group**) is the symmetry group of a rectangle, which has four elements, the identity, the vertical reflection, the horizontal reflection, and a 180 degree rotation, as shown in Figure 4. The rectangle symmetry group has the addition table shown in Figure 4, Panel A.

The symbol ⊕ means "followed by". For example, v ⊕ h = r means that a vertical reflection followed by a horizontal reflection is the same, as if it had rotated the rectangle by 180 degrees.

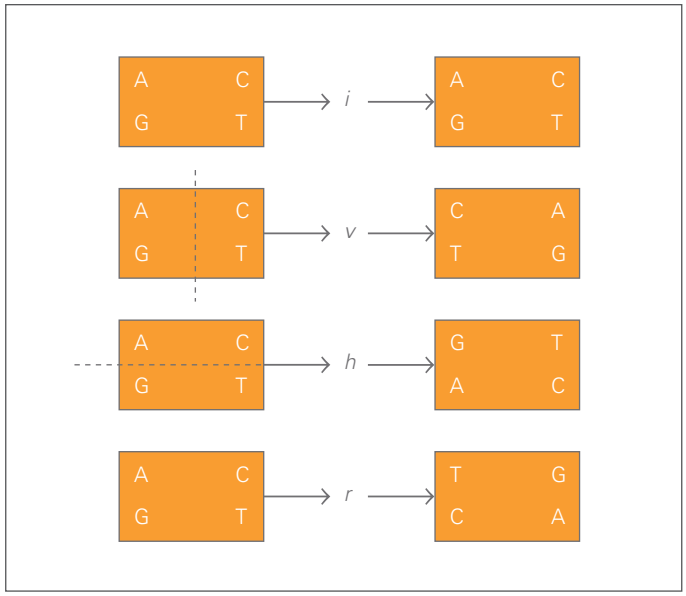

Figure 5: Using the Klein four-group to Obtain the Code for Strings of Colors as Transformations of Bases. Rectangle symmetries *i, v, h,* and *r* transform bases exactly like color operators 0, 1, 2, and 3 respectively.

The set of color operations with functional composition that were discussed above is isomorphic to the Klein four-group. This can be observed by labeling the corners of a rectangle with the bases and witnessing their rearrangements (transformations), as shown in Figure 5. The identity leaves the bases unchanged, exactly like color 0. The vertical reflection swaps A with C, and G with T, exactly like color 1. The horizontal reflection swaps A with G, and C with T, exactly like color 2; and the 180 degree rotation swaps A with T, and C with G, exactly like color 3. An addition table can be created for colors from the addition table for rectangle symmetries simply by substituting i, v, h, and r with 0, 1, 2, and 3 respectively, as in Figure 4, Panel B. The symbol ⊕ means "followed by" in this context too, but it is usually more convenient to leave it out in the written expressions. For example, instead of 3⊕2⊕1⊕0⊕2⊕3⊕0⊕2⊕2, just write 321023022, and A321023022 = C means the same as f2(f2(f0(f3(f2(f0(f1(f2(f3(A)))))))) = C, the former being somewhat easier to read, write, and parse.

Properties of Groups

A group (**http://en.wikipedia.org/wiki/Group_theory**) is a set of elements (colors, in our case) with an operator ⊕ that satisfies the following four properties:

1) Closure: If a and b are elements, then a ⊕ b is also an element.
2) Associative: $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
3) Identity: There is an element *i* such that $i \oplus a = a \oplus i = a$.
4) Inverse: For every element a, there is an element *a*-1, such that $a \oplus a\text{-}1 = a\text{-}1 \oplus a = i$.

In addition, the Klein four-group is also Abelian, which is to say that it is

5) Commutative: $a \oplus b = b \oplus a$.

In this group of color calls, '0' is the identity element and every element is its own inverse (the diagonal of Figure 4, Panel B are all 0's). It should be noted that except for the identity color '0', any color composed with any other color is the third color. These properties can be used when designing protocols, writing programs, and proving correctness.

Applications to SOLiD Sequencing Events

Using the theoretical methods discussed above, here are a few examples on how to analyze typical SOLiD resequencing events. Each of the next few sections analyzes common resequencing events by using an example of a nucleotide sequence read, the corresponding reference, and their color-string representations. In general, a color string that does NOT satisfy these requirements is said to be "invalid".

Mismatched Leading Base

In this application, a given color read and its alignment with a reference is shown below. If all *colors* match, but the leading *base* does not, then *ALL* decoded bases will be mismatches. This is observed in the example below.

```
  A C G T A C G T C C G T A C
A 1 3 1 3 1 3 1 2 0 3 1 3 1
C 1 3 1 3 1 3 1 2 0 3 1 3 1
  C A T G C A T G A A T G C A
```

The color code group properties can be used to prove this will always be the case. Begin with just the alignment of color reads:

A 1 3 1 3 1 3 1 2 0 3 1 3 1
C 1 3 1 3 1 3 1 2 0 3 1 3 1

For all positions *i*, the reference base at *i* is not equal to the read base at *i*. In considering an arbitrary position *i*, it has already been shown that the base in reference position *i* is A*b*, where *b* is the sum of the reference colors in positions 1 through *i*, using the addition table in Figure 4, Panel B. Similarly the base in read position *i* is C*d*, where *d* is the sum of read colors in positions 1 through *i*. However, *b* must be equal to *d* because the read colors are the same as the reference colors. Therefore, by considering color code requirement 2, A*b* ≠ C*b*, *which shows all bases must be mismatches.*

An Isolated Single-Base Variant

A single-base variant is a read base that differs from its aligned reference base. The variant is said to be isolated if it is not adjacent to a variant on either side. For example, the dot marks the isolated single base variant in the alignment below.

```
              1 3 1 3 1 3 1 3 1 3 1 3 1
Reference: A C G T A C G T A C G T A C
           | | | | | | | • | | | | | |
Read:      A C G T A C G A A C G T A C
              1 3 1 3 1 3 2 0 1 3 1 3 1
```

To analyze this case, let's focus on the variant and introduce variables, as shown in the alignment below.

```
            u v
Reference: B D F
           | • |
Read:      B E F
            w x
```

Here D ≠ E are mismatched bases, and they are isolated because they are flanked by matched bases B and F. In order for colors u, v, w, and x to be consistent with this variant, they must satisfy the following properties:

1. u ≠ w because color (BD) ≠ color (BE) by Code Requirement 2.
2. uv = wx because both colors uv and wx transform base B to base F, so that uv = color (BF) = wx.
3. v ≠ x follows from the first two requirements. This does not need to be tested since the first two are true.

Two Adjacent Variants

In the previous application, both relevant color positions were mismatches. The two examples in this section show that some colors in other kinds of resequencing events, although otherwise constrained, may result in matches or mismatches. Therefore, when annotating color reads, it is not sufficient to annotate only colors that are mismatches. Here is an example where all relevant positions are color mismatches:

```
              1 3 1 3 1 3 1 3 1 3 1 3 1
Reference: A C G T A C G T A C G T A C
           | | | | | | • • | | | | | |
Read:      A C G T A C A A A C G T A C
              1 3 1 3 1 1 0 0 1 3 1 3 1
```

Analysis:

```
            u v w
Reference: B D E F
           | • • |
Read:      B G H F
            x y z
```

Properties:

1. u ≠ x by Code Requirement 2.
2. uv ≠ xy because they must transform B to *different* bases E and H.
3. uvw = xyz because both uvw and xyz must transform B to F.
4. v and y could be equal or they could be different.

There are two additional interesting properties that follow from the properties above, therefore, there is no reason to test for them when identifying two-base variants above.

5. vw ≠ yz because they must transform *different* bases D and G to F.
6. w ≠ z because they must transform *different* bases E and H to F.

Here is another example of two adjacent variants resulting in three color positions in which only the outer two are mismatches.

```
              1 3 1 3 1 3 1 3 1 3 1 3 1
Reference: A C G T A C G T A C G T A C
           | | | | | | • • | | | | | |
Read:      A C G T A C C A A C G T A C
              1 3 1 3 1 0 1 0 1 3 1 3 1
```

Even though the center two color positions, both with color 1, are not mismatches, the analysis above still applies, and this new example satisfies all of the above properties.

Three Adjacent Variants

Here is an example of three adjacent variants.

```
          1 3 1 3 1 3 1 3 1 3 1 3 1
Reference: A C G T A C G T A C G T A C
           | | | | | • • • | | | | | |
Read:      A C G T A G C A A C G T A C
          1 3 1 3 2 3 1 0 1 3 1 3 1
```

This example has *two* single-color mismatches isolated by two color matches. In the analysis below, the B_i are bases and the c_i are colors:

```
            c1  c2  c3  c4
Reference: B1  B2  B3  B4  B5
           |   •   •   •   |
Read:      B1  B6  B7  B8  B5
            c5  c6  c7  c8
```

Properties:

1. $c_1 \neq c_5$
2. $c_1c_2 \neq c_5c_6$
3. $c_1c_2c_3 \neq c_5c_6c_7$
4. $c_1c_2c_3c_4 = c_5c_6c_7c_8$

Insertion and Deletions (Indels)

In these examples, a base deleted from the read is depicted as a '—' in the read. Similarly, a base inserted into the read is depicted as a '—' in the reference. It should be noted that the reference would never be stored with a '—' in those places, nor does the SOLiD™ System ever call a '—' for any of its colors. The '—' would be generated only by aligning the read with the reference. For example, here is a single-base deletion:

```
          1 3 1 3 1 3 1 3 1 3 1 3 1
Reference: A C G T A C G T A C G T A C
           | | | | | | • | | | | | |
Read:      A C G T A C — T A C G T A C
          1 3 1 3 1 2 — 3 1 3 1 3 1
```

The analysis must explain two situations, which it does:

```
            u v           u v
Reference: B D E         B D E
           | • |  and    | • |
Read:      B - E         B -
             - w           w -
```

Properties:

1. uv = w because both colors uv and w must transform base B to base E. A single base deletion causes two elementary colors, *u* and *v*, to be replaced by one, *w*. In the above deletion, for example, 31 = 2, as is easily confirmed by the addition table.
2. Property 1 is a sufficient test, but it may be of interest that the addition table implies that either u, v, and w are all different, or one is 0 and the other two are identical. Property 1 applies also for multiple bases.

For example, here is a three base insertion:

```
          1 3 1 3 1 - 3 - - 1 3 1 3 1 3 1
Reference: A C G T A C - - - G T A C G T A C
           | | | | | | • • • | | | | | | |
Read:      A C G T A C T A G G T A C G T A C
          1 3 1 3 1 2 3 2 0 1 3 1 3 1 3 1
```

This alignment shows the 3 in the insertion aligning with a 3 in the reference, even though these two have nothing to do with one another; the 3 in the reference encodes CG, whose bases bracket the insertion, and the insertion has a TA, which is coincidently also encoded with color 3.

Analysis:

```
             u
Reference: B - - - H
           | • • • |
Read:      B D E F H
             v w x y
```

The alignment above shows the reference color u aligned with read color v, but in fact might align with color w, x, or y instead. The following property holds in all cases: u = vwxy. One elementary color, u, in the reference is replaced by four, c_1, c_2, c_3, and c_4, in the read.

Correcting Two Common Misunderstandings

A cursory overview of the SOLiD 2 base color coding system may lead to some common misunderstandings. The first is that **an isolated color mismatch must always correspond to a sequencing error**. The second is that certain (3/4 of the total) **isolated two-color changes always correspond to errors**. These misunderstandings do not actually appear in print, but there is public information that can be easily misread that way if taken out of context.

In this section, two examples are presented that discuss these common misunderstandings. The first involves an isolated pair of adjacent base variants and addresses only the isolated color change problem. The second example involves an isolated cluster of three adjacent base changes. The second example has both an isolated color change and an "impossible" two color change.

Counter Example to "an isolated color change always corresponds to a sequencing error."

```
          2 0 1 3 1 2 0
Reference: G A A C G T C C
           | | | • • | | |
Read:      G A A G C T C C
          2 0 2 3 2 2 0
```

This counter example shows two single-color mismatches, both 1→2, isolated from one another by a color match. It is easy to verify that this counter example does indeed satisfy the requirements for a two-base variant: 1 ≠ 2, 13 = 2 ≠ 1 = 23, and 131 = 3 = 232.

Counter Example for "impossible two position changes always correspond to measurement errors."

```
          0  1  2  0  3  0
Reference: A  A  C  T  T  A  A
           |  |  •  •  •  |  |
Read:      A  A  T  G  G  A  A
           0  3  1  0  2  0
```

The example shows a "forbidden" (i.e., impossible adjacent mismatches when only a single base change is present) two-color change, 12 → 31, isolated by a color match (0 → 0) from a single-color change (3 → 2). Both "impossible" color changes are nevertheless perfectly consistent with a three-base variant, in particular, the one shown above.

Again, it is easy to verify that the colors satisfy the requirements for a three-base variant: 1≠3, 12≠31, 120≠310, and 1203 = 3102.

A clarification of the misunderstanding follows easily by understanding that they are equivalent to assuming that the only base variants permitted in sequences are isolated single-base substitutions.

Annotating Alignment Mismatches in the SOLiD™ System

From the previous discussion, it is clear that when a SOLiD read is aligned to a reference sequence, only a subset of possible mismatches represent underlying sequence variations. It is therefore important to identify mismatches that are candidate sequence variations and to distinguish them from other patterns that are the result of sequencing error. This information can then be used to correct errors from aligned reads, resulting in a significant improvement in the accuracy of the system, and to pass putative sequence variation to downstream algorithms for the assessment of genotype calls and identification of new alleles in the sample. These annotations are typically conveyed in files that describe the attributes of sequence reads, such as the SOLiD GFF file.

The SOLiD GFF is an important file format for the SOLiD System. In essence, a file of this type is a list of reads, each with several attributes. One of those attributes is the 's', or annotations attribute. Here is the definition of the annotations attribute of the SOLiD GFF v 0.2 file:

> This is a comma-separated string representing annotations on the sequence. The format is '{char}{position}' where {char} is a character representing the type of annotation (typically a formatting request to visualization software) and {position} is the position of this annotation in the read. The position is 1-based on the string recorded in the 'g' attribute. That is, the prepended base has position 1, and the first color has position 2. For example, "a5, g7, g8" means "format the color call at position 5 gray, format call 7 green, and format call 8 green". The SOLiD System follows this convention:

a (Gray) is an *isolated mismatch*; it is a mismatch and neither the color-call on its left nor the color-call on its right is a mismatch,

g (Green) is a *valid adjacent mismatch*; it is a mismatch and this mismatch, together with the adjacent mismatch on its left or right, could correspond to an isolated SNP,

y (Yellow) is a color call that is consistent with an isolated *two*-base change. In general, these will be mismatches, but a conserved color between two mismatches is also a possibility.

r (Red) is a color call that is consistent with an isolated *three*-base change.

b (Blue) is an *invalid adjacent mismatch*; it is any other mismatch.

An efficient algorithm for calculating these annotations, given an alignment of the color reads with a corresponding color reference, can be derived from the theory described previously. To begin, the following theorem correlates the applications in this section to SOLiD sequencing events.

Theorem 1:

Let $c = c_1c_2c_3 \ldots c_k$ be a k-color substring of a read aligned with the corresponding color reference $r = r_1r_2r_3 \ldots r_k$. Then c encodes an isolated (k-1)-base change if and only if the base position preceding c is not a variant, and the following two equations hold under the Color Addition Table (Figure 4, Panel B):

Equation 1:

$$\sum_{j=1}^{k} c_j = \sum_{j=1}^{k} r_j$$

Equation 2:

For all i from 1 to k-1

$$\sum_{j=1}^{i} c_j \neq \sum_{j=1}^{i} r_j$$

Proof: For the forward direction, suppose that c encodes an isolated k-1 base change.

```
                 r1  r2     • • • rk-1  rk
Reference: B0  B1  B2  • • •  Bk-1  Bk
           |   •   •   • • •   •     |
Read:      B0  D1  D2  • • •  Dk-1  Bk
                 c1  c2     • • • ck-1  ck
```

Then the base B_0 preceding c is not a variant because it must serve to isolate the k-1 base change. Similarly, the base B_k following c is also not a variant because it too must serve to isolate the base change. Equation 1 and 2 are derived by using the color addition properties.

$$\sum_{j=1}^{k} c_j = color(B_0B_k) \quad \text{and} \quad \sum_{j=1}^{k} r_j = color(B_0B_k)$$

Equation 2 stays the same, because all bases between B_0 to B_k are variants.

To prove the reverse direction, suppose that the base B_0 preceding color c is not a variant, as shown below, and that Equations 1 and 2 hold.

$$\begin{array}{lccccccccc} & & r_1 & r_2 & & \cdots & r_{k-1} & r_k & \\ \text{Reference:} & B_0 & B_1 & B_2 & \cdots & & B_{k-1} & B_k & \\ & | & \cdot & \cdot & \cdots & \cdot & & | & \\ \text{Read:} & B_0 & D_1 & D_2 & \cdots & & D_{k-1} & D_k & \\ & & c_1 & c_2 & & \cdots & c_{k-1} & c_k & \end{array}$$

Then Equation 1 implies that $D_k = B_k$, as follows:

$$D_k = B_0 \sum_{j-1}^{k} c_j = B_0 \sum_{j-1}^{k} r_j = B_k$$

Bases D_1 through D_{k-1} are therefore isolated by matched bases B_0 and B_k. Furthermore, all isolated bases are variants that are $D_i \neq B_i$ for all i between 1 and k-1, which follows with the help of Equation 2:

$$D_i = \sum_{j-1}^{i} c_j \neq \sum_{j-1}^{i} r_j = B_i$$

The equation above completes this proof. If a substring of colors is consistent with a k-base change, assume the leading base is not a variant and apply Theorem 1. To annotate color calls, note that a color call is consistent with a k-base change if it is part of a color string that is consistent with a k-base change.

Theorem 2: A color call that is consistent with a k-base change is not also consistent with an m-base change for $m \neq k$.

Proof: Assume the opposite, so that there is a color call c_i that is part of a color string c that is consistent with a k-base change, and also part of a color string d that is consistent with an m-base change. There are two substrings of bases, of differing lengths k+2 and m+2, where the outer two bases are matches and the inner k and m form the isolated variants. Assume without loss of generality that the k-base sequence overlaps the m-base sequence, at color c_i, "on the left". The extreme case is sketched below, in which a matched base is a '1' and a mismatched base is an 'x'. Recall that color calls appear only between bases.

```
1xxx1        k-base mismatches on the left
   1xxxx1    m-base mismatches
```

The first base of the m-base sequence, a match, coincides with a mismatch in the k-base sequence, which contradicts the fact that a base cannot simultaneously match and mismatch the reference. This contradiction proves the theorem.

The Algorithm

The pseudo-code function getCompatibility, shown here, returns m if the color call at position j is compatible with an isolated group of m mismatched bases (variants), where m is between 0 and a preset maximum acceptable number M. For readability, this pseudo-code ignores read "edge effects".

By Theorem 1, position j is compatible with m isolated variants at line ***break***. By Theorem 2, there is no need to continue searching, because j will not be compatible with any other number. Note the efficiency of this algorithm: if j is compatible with m variants, the algorithm makes the least number of tests possible before breaking from the loop. On the other hand, all the tests that it does make are required by Theorem 1 because if it skips any test, an "adversary" could devise an input on which the algorithm would give the wrong answer. See Appendix 2 for an implementation in Java. Figure 6 shows a small portion of a visualization of a GFF file annotated with this method.

Algorithm: getCompatability
Input: Read colors *C*, Reference colors *R*, Query position *j*, Maximum acceptable variants *M*
Output: Number of compatible isolated base variants m.

```
Method:
  //sumCjm is the group theoretic sum of m read colors c_i starting at i=j.
  //sumRjm is the group theoretic sum of m read colors r_i starting at i=j.
Initialize: m = sumCjm = sumDjm = 0; n = size(c)

  while m ≤ M, do
  {
     sumCjm = additionTable[sumCjm][cj];
     sumRjm = additionTable[sumRjm][dj];
     if (sumCjm == sumRjm)
       break; // Substring s = cj…c_{j+m} is compatible with an isolated group of m
       adjacent variants.
     m = m + 1
  } // for m

  // If the loop terminates with m = M+1, then j is not compatible with M or less
  isolated variants.
  if (m > M) return "Incompatible"
  else return m
```

Conclusion

The existing 2 base color code for SOLiD sequencing follows from a small set of requirements. This color code is essentially the only one that satisfies these requirements. If colors are treated as transformers of bases, these requirements lead to a formulation of color combinations as the well-known Klein four-group. The group theoretic properties lead to efficient and systematic methods for annotating color reads from the SOLiD™ System. In particular, they allow for a simple Java implementation that annotates color-calls with any given number of DNA variants.

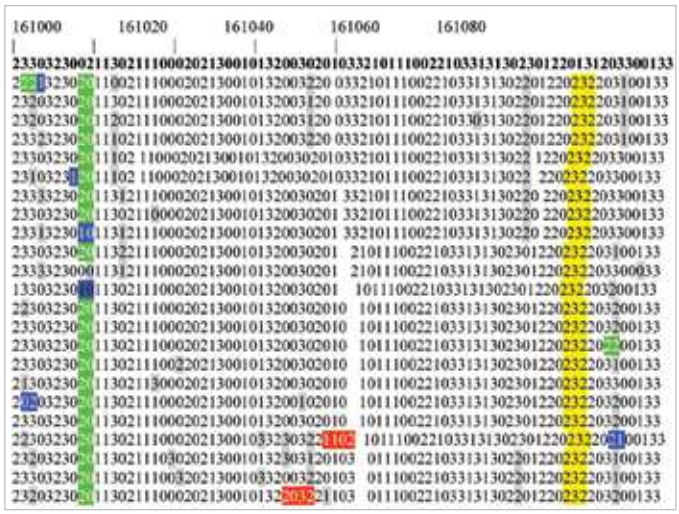

Figure 6. Visualization of an Annotated GFF file Using the getCompatibility Algorithm.

References

1. Costa, Gina, et al. Applications of Next Generation Sequencing Using Stepwise Cycled-Ligation, http://marketing.appliedbiosystems.com/images/Product/Solid_Knowledge/PDF/ASHG_2007_2791.pdf
2. Peckham, Heather E. et al. SOLiD Sequencing and 2 Base Encoding, http://marketing.appliedbiosystems.com/images/Product/Solid_Knowledge/PDF/ASHG_2007_2624.pdf
3. Rhodes M. SOLiD System Data 2 Base Encoding, Webinar, http://appliedbiosystems.cnpg.com/lsca/webinar/rhodes/2be/20070618/

		Second Base			
		A	C	G	T
First Base	A	2	3	0	1
	C	3	2	1	0
	G	0	1	2	3
	T	1	0	3	2

Figure 7. A permuted color addition table.

Appendix 1: Alternative 2 Base Color Coding Schemes

This appendix explores alternative color coding schemes that exist, and their effects on subsequent group properties.

Maintaining the Requirements

There is essentially only one coding scheme that satisfies all the *requirements*. Once the first row has been decided, the requirements determine the rest of the table, as shown in Figure 3. Each of the 4! = 24 ways to lay down the first row (given requirements 1 and 2) corresponds to a "different" table. But these are all isomorphic to one another. Even though there can only be one table, the colors can be permuted to obtain 23 other tables. For example, the permutation:

$0 \Rightarrow 2$
$1 \Rightarrow 3$
$2 \Rightarrow 0$
$3 \Rightarrow 1$

generates the table in Figure 7.

Relaxing the Constraints

Different coding schemes can be generated by relaxing the constraints. Unfortunately, these alternative schemes are not guaranteed to satisfy the group theoretic properties that are needed for annotation, error detection, and correction. Here are four of the requirements:

For all bases *b, d, e* in B:

1. The available colors are 0, 1, 2, and 3:
   color (*bd*) ∈ {0,1,2,3}.
2. Two different di-bases that nevertheless have the same first base get *different* colors:
   color (*bd*) ≠ color (*be*) if *d* ≠ *e*.
   For example, color (AC) ≠ color (AG).
3. A di-base and its reverse get the *same* color:
   color (*bd*) = color (*db*)
   For example, color(AC) = color(CA).
4. Monodibases get the *same* color:
   color (*bb*) = color (*dd*)
   That is, color (AA) = color (CC) = color (GG) = color (TT).

		Second Base			
		A	C	G	T
First Base	A	0	1	2	3
	C	1	3	0	2
	G	2	0	3	1
	T	3	2	1	0

Panel A

		Second Base			
		A	C	G	T
First Base	A	0	1	2	3
	C	1	0	3	2
	G	3	2	0	1
	T	2	3	1	0

Panel B

Figure 8. Dropping Color Code Requirements Will Not Satisfy All Group Properties. (A) A color code that satisfies only requirements 1, 2, and 3, now does not allow 01 to behave like a single color. **(B)** A color code that satisfies only requirements 1, 2, and 4, now does not allow 23 to behave like a single color.

Can requirement 4 be dropped? One such table is shown in Figure 8, Panel A, but it no longer satisfies the group properties. For example 01 does not behave like a single color anymore: A01 = C and A1 = C, but C01 = T and C2 = T. So color 0⊕1 behaves like color 1 for A, but it behaves like color 2 for C.

Can we drop just requirement 3? Such a table is shown in Figure 8, Panel B, but it no longer satisfies the group properties. For example 23 does not behave like a single color anymore: A23 = C and A1 = C, but C23 = C and C0 = C. So color 2⊕3 behaves like color 1 for A, but it behaves like color 0 for C.

Appendix 2: A Java Implementation of getCompatibility

The Java function below, getCompatibility, implements the pseudo-code in the Section *The Algorithm*.

```
private int getCompatability( String readColors, String refColors, int j )
 {
  //sumCjm is the group theoretic sum of ref color ci from i=j to j+m.
  int m = 0; // Number of color 'mismatches'.
  int sumCjm = 0;
  int sumDjm = 0;
  int n = Math.min(readColors.length(), refColors.length());
  int numRemaining = n - j - 1; // The number of positions past j.
  // maxM is the largest block size m to check.
  int maxM = (MAX_ADJACENT_VARIANTS < numRemaining)
        ? MAX_ADJACENT_VARIANTS : numRemaining;

  for (m=0; m<=maxM; m++)
  {
   int cj = Character.getNumericValue(refColors.charAt(j+m));
   int dj = Character.getNumericValue(readColors.charAt(j+m));
   // Add in the new colors.
   // Assume missing data if either is out of range.
   if (0 <= cj && cj <=3 && 0 <= dj && dj <= 3) // in range.
   {
    sumCjm = addColors[sumCjm][cj];
    sumDjm = addColors[sumDjm][dj];
    if (sumCjm == sumDjm)
     break; // Compatible with isolated group of m adjacent variants.
   }
   else // out of range. Terminate this test as if we ran over maxN.
   {
    m = maxM + 1;
   }
  } // for m

  // Return m if it is in range. Otherwise, 0 means incompatible and isolated,
  // and -1 means incompatible but not isolated.
  if (m < 1 || m > maxM) { m = isIsolated(j, readColors, refColors) ? 0 : -1; }

  return m;
 } // getCompatability()
```

Printed in the USA. 07/2010 Publication 139WP01-02 CO13982

applied biosystems™
by life technologies™

Headquarters
850 Lincoln Centre Drive | Foster City, CA 94404 USA
Phone 650.638.5800 | Toll Free 800.345.5224
www.appliedbiosystems.com

International Sales
For our office locations please call the division headquarters or refer to our Web site at
www.appliedbiosystems.com/about/offices.cfm