The FASTG Format Specification (v1.00)

An expressive representation for genome assemblies

The FASTG Format Specification Working Group

December 12, 2012

1. Introduction

This document introduces a prototype format for faithfully representing genome assemblies in the face of allelic polymorphism and assembly uncertainty. It is called FASTG, like FASTA, but the G stands for 'graph'.[1]

As motivation for this new representation, consider the traditional *linear* representation of genome assemblies, which exhibits each scaffold as a FASTA record. Key advantages of this representation are that its linearity matches the linearity of the genome, and that it naturally provides coordinates, thus facilitating computation and display. It is also very easy to parse, easy for end-users to read and interpret, permits flexible annotations with custom fields, and perhaps most importantly has a universe of tools built around it.

This traditional linear representation, however, has one deep flaw – it has very limited capacity to express branching that can arise from allelic polymorphism, intrinsic limitations of the data, or even limitations of the algorithms used to assemble them. This inability to represent branching and retain intrinsic ambiguities has many negative consequences, including the forced introduction of errors (where instead uncertainty could be shown), and typically, the failure to express information about polymorphism that is contained in the raw data and typically available internal to assembly algorithms. In short, an 'honest' assembly would tell you if it didn't know the answer, or if more than one answer was possible, and to do this a representation capable of branching would be required.

This is not a new concept, and indeed the traditional assembly representation had two mechanisms for representing branching and uncertainty: (1) ambiguous base codes, according to the IUPAC scheme, and (2) base quality scores (via an ancillary quality score file, in one-to-one correspondence with the FASTA file). Both are limited in their expressive capability, and while (1) is exceptionally straightforward and easy to understand, the interpretation of (2) is less clear, particularly in regard to indels. Finding an appropriate replacement will involve a balance between expressive capability and complexity.

A very general solution would be to represent assemblies as large, all-inclusive graphs that capture the information that the assembly algorithm has extracted from an inaccurate and redundant raw dataset. This solution, however, would lose the key advantages of the traditional linear representation, as the

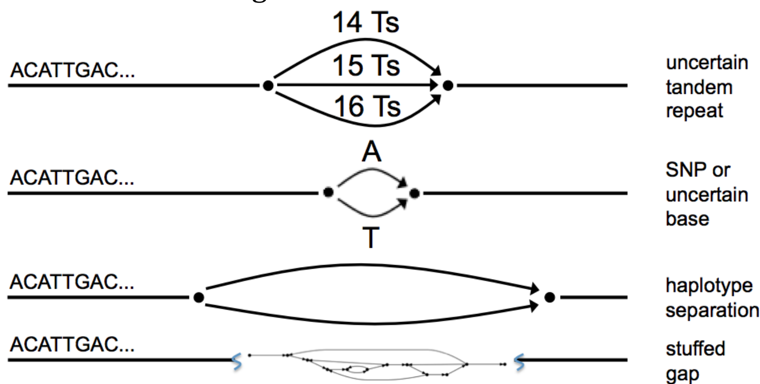
---

[1] FASTA originally stood for a fast algorithm that could align any letters (not just nucleotides or protein symbols), but came to represent the file format inputted to that algorithm [1], [2], http://blast.ncbi.nlm.nih.gov/blastcgihelp.shtml, http://en.wikipedia.org/wiki/FASTA_format.

resulting all-inclusive graphs can be complex and lose many of the advantages in the traditional linear representation that we wish to retain.

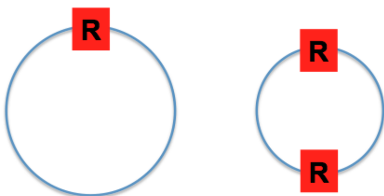
Here we propose instead a hybrid approach that preserves local linearity (and its associated benefits) where possible but also captures additional nonlinear features of the assembly. In essence, the FASTG format offers 'FASTA-plus'. Any assembly in this format can be canonically converted into FASTA, albeit with loss of information, and this FASTA could have a coordinate-based markup that precisely describes the lost information. Such a canonical conversion to conventional FASTA format seems to be a prerequisite for any format to be widely adopted and generally useful.

Graphical features that are needed may be roughly characterized as 'local' or 'global'. For example, if we know that a particular base is a SNP, then that's a local feature. The most common local features look like one of the following:



and so we've designed the format to handle these types of events easily. Importantly, the first three all preserve linearity relative to the genome by capturing variation in the form of a directed acyclic graph, whereas the fourth (§4) allows cycles, but in highly controlled context.

While many assemblies can be represented without global features at all, some are unavoidable. Consider for example the genome of the bacterium *Rhodobacter sphaeroides*. It has two circular chromosomes, containing three instances of a 5.2 kb perfect repeat R, all in the same orientation.



Now supposing that one's assembly input data includes long links that can bridge R, it is in principle possible to complete resolve its genome, yielding two circular chromosomes,

which would be represented by two fasta records, one per chromosome. On the other hand, suppose that one does not have such long links. Then the best one can do is to represent the assembly as a graph having four edges

which would correspond to four fasta records.

Note that a 'perfect' assembly would have essentially no local or global structure, and could be represented by one FASTG record per physical chromosome (thus with homologous chromosomes represented separately). Additional structure would only be needed to capture the difference between linear and circular chromosomes. In particular, in such a 'perfect' assembly, 'repetitive' sequences would appear multiple times in the assembly, in accordance with their actual copy number and local sequence variations. For some small genomes, such perfect assemblies could in principle be achieved with existing data, and even for large genomes, it makes sense to strive for assemblies that are as 'perfect' as possible. Fully 'perfect' assemblies of large genomes are unlikely to be produced in the near future, as they imply (a) complete phasing of all allelic variations, which requires a network of paired-end information at many length scales, or sequence from other family members, and (b) methods for disentangling repetitive sequences with complex nested or tandem structures that are inaccessible, even theoretically, with existing data types.

Below we first present some intuitive examples, without defining all the terms in advance, and then dive deeper into the formal definition of the terms. We first note the following design principles:

- In this format, it is a rule that every construct can be converted into FASTA. Thus for example `[1:alt|A,T,AA]` is an example of an 'ambiguity' string representing either A or T or AA, and by definition the *first* alternative (in this case A) would be used in the canonical conversion to FASTA.
- The canonical representation of each FASTG construct is given just prior to the construct itself. In the above example `[1:alt|A,T,AA]` would be preceded by A. (See next section for an example.) As a convenience, the length of this canonical sequence is encoded as the first field in the FASTG construct. We adapted these conventions so that FASTG could be converted to its canonical FASTA form using only a few lines of code.
- This format uses 'inline' representation, in which constructs appear inline as insertions in a FASTA file. In an alternate version of the format, the constructs would appear instead in a coordinate-based markup that accompanies the FASTA as an ancillary file. This markup format is simply the FASTG constructs plus the base offset at which they occur and is described later. We anticipate that other versions of markup for FASTG may be needed but do not attempt to specify them here.
- The format includes directed graphs in which each edge represents a DNA sequence (as described in §6). A path through the graph is to be interpreted as corresponding to the *concatenation* of these sequences. We allow for reverse complements, so as to allow for the representation of complex inversion events. Here we have borrowed from the notion of connection, as in the SeQuence Graph (SQG) file format (`https://github.com/benedictpaten/pysqg`).
- The format is hierarchical, thus allowing a distinction between global and local structure, and effectively sequestering graph structure, so as to present assemblies in as linear and FASTA-like a form as possible. At least three levels might be needed. For example, this would be needed to describe an assembly having global structure, and within that, haplotypes separated over a fairly long range, and within each haplotype branch, uncertainty about particular bases.

2. A toy example

Consider the following simple representation of an assembly of two chromosomes that includes a gap, a SNP, and a tandem repeat embedded within otherwise known sequence:

```
#FASTG:begin;
#FASTG:version=1.0:assembly_name="tiny example";
>chr1:chr1;
ACGANNNNN[5:gap:size=(5,4..6)]CAGGC[1:alt:allele|C,G]TATACG
>chr2;
```

```
ACATACGCATATATATATATATATATATAT[20:tandem:size=(10,8..12)|AT]TCAGGCA[1:alt|A,
T,TT]GGAC
#FASTG:end;
```

This is a complete FASTG file, with required 'begin' and 'end' lines, however which for brevity, we will drop from subsequent examples. In a FASTG file, all white space (blanks, tabs and newlines) is ignored (except in double-quoted 'literal' strings), so we are free to insert it wherever we like to enhance readability.

The example represents an assembly having two scaffolds named `chr1` and `chr2`. Here are some of its features:

- The expression `[5:gap:size=(5,4..6)]` represents a gap of between 4 and 6 bases, with 'default' 5. Since the default is 5, the preceding canonical sequence is NNNNN.
- The expression `[1:alt:allele|C,G]` represents an 'ambiguity': it is either C or G. The canonical sequence is C, and so the size of the FASTG construct is 1. The `allele` property specifies that both C and G are present in the sample at the given locus, and in equal proportions. Typically this would be because two homologous chromosomes are represented by a single scaffold in the assembly. Alternatively, a single scaffold could collapse two highly similar regions that are not homologous.
- The expression `[20:tandem:size=(10,8..12)|AT]` represents a run of between 8 and 12 ATs. Because 10 is listed first, it is regarded as the 'primary' representation and therefore the canonical sequence is 10 ATs. In this case no information is provided as to the relative likelihood of the different run lengths, however, as is always the case for such ambiguous expressions, it is 'guaranteed' that at least one of the runs appears in the sample at the given locus. Note that because of polymerase slippage, ambiguities of this type are very common, and indeed may correspond to (1) the primary error mode in input data, and (2) a common polymorphism in complex genomes. With existing whole-genome shotgun data types it is often impossible to distinguish between these two situations.
- The expression `[1:alt|A,T,TT]` asserts that at least one of A, T or TT is present in the sample at the given locus. In this situation, the canonical sequence is defined to be A, and so the size of the FASTG construct is given as 1.

Importantly, in this format there is always a fully specified conversion of an assembly into 'pure' FASTA. Every expression comes equipped with a mechanism for picking a particular sequence wherever there is an option, which is typically the first in a list. Here is the mapping to FASTA for each of the above four expressions:

```
[5:gap:size=(5,4..6)]             NNNNN
[1:alt:allele|C,G]                C
[20:tandem:size=(10,8..12)|AT]    ATATATATATATATATATAT
[1:alt|A,T,TT]                    A
```

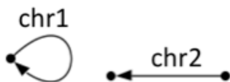
These canonical sequences are always given just prior to the FASTG constructs. Simply stripping out the constructs gives the corresponding FASTA, with loss of information. In the example below the canonical sequence is in red, whilst the FASTG construct it represents is in **bold**. (color added for emphasis):

```
>chr1:chr1;
ACGANNNNN[5:gap:size=(5,4..6)]CAGGC[1:alt:allele|C,G]TATACG
>chr2;
ACATACGCATATATATATATATATATATAT[20:tandem:size=(10,8..12)|AT]TCAGGCA[1:alt|A,
T,TT]GGAC
```

Becomes:

```
>chr1:chr1;
ACGANNNNNCAGGCTATACG
>chr2;
ACATACGCATATATATATATATATATATATTCAGGCAGGAC
```

There is one more piece of information encoded in the assembly via the FASTG 'header line'. The notation `chr1:chr1` declares that `chr1` is adjacent to itself. In other words, it is a circular edge. On the other hand, `chr2` does not has no adjacencies, indicating that it does not connect to anything else. In short, a graphical view of the assembly is:



which exhibits one little bit of global structure, namely the circularity of `chr1`.

3. Representation of haplotypes

Depending on the properties of a genome and data from it, it may be possible to separate haplotypes over a relatively long range. FASTG is set up to support this. For example, an `alt` expression can be used to describe a haplotype 'bubble' in the case where the individual haplotypes are completely known, *e.g.*

```
[30:alt|CATGGCAACCTAGCA CCTATACGAGCTTAC,
        CAAGGCAACCAAGCAACCTATACGAGCTTAC]
```

FASTG also allows for cases where the haplotypes themselves have features (such as uncertain bases or gaps), using the `digraph` expression, e.g.

```
[30:digraph:path=(a),begin=(a,b),end=(a,b) |
      >a;
      CATGGCAACCTAGCA CCTATACGA  G[1:alt|G,C] CTTAC
      >b;
      CAAGGCAACCAAGCAACCTATACGA  G            CTTAC   ]
```

This represents a very simple graph, having two 'parallel' edges. The notation `path=(a)` specifies that the canonical path through the graph is given by the first edge. This example also illustrates the nested nature of FASTG, where it is possible to encode a FASTG construct within another FASTG construct.

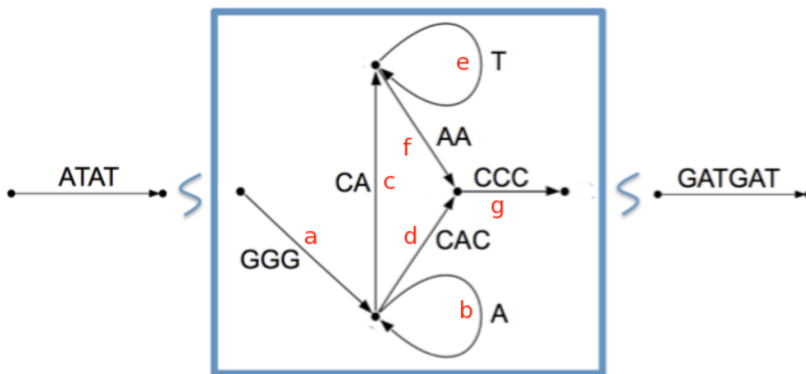
The entry point to the graph is given by the property `begin`, which contains a list of one or more edges. Similarly the property `end` gives the exit point of the graph, again as a list of edges. These provide the linking information between the edges in the subgraph and the surrounding sequence. For this simple case where there are only parallel edges, the `begin` and `end` properties may be omitted.

4. A stuffed gap

Consider the following assembly, which has been generously padded with white space to enhance readability:

```
>Z;
ATAT NNNNNNNNNNNNNNN [13:gap:size=(13,10..35), begin=a, end=g |
                      >a:b,c,d;
                      GGG
                      >b:b,c,d;
                      A
                      >c:e,f;
                      CA
                      >d:g;
                      CAC
                      >e:e,f;
                      T
                      >f:g;
                      AA
                      >g;
                      CCC          ]    GATGAT
```

This assembly contains a single scaffold `Z`, having one gap. However it is not an ordinary gap: it is a *stuffed* gap, with a mini-assembly-graph inside the gap, as in the following picture where the edge names are given in red:



The assembly claims that the true genome sequence for the gap is represented by some path through the boxed 'gap' graph of length between 10 and 35. This is actually a very common situation. From an assembly, one can often exhibit a graph corresponding to what should be inside a gap, but either due to limitations of the data or other algorithmic reasons, be unable to resolve the exact sequence. Since we don't know the most likely path, the canonical sequence is give as a gap of 13 bases – represented as 13 Ns.

Such incomplete information can be exceptionally valuable. For example, we have seen cases where a disease-causing frameshift mutation is hiding in a (stuffed) gap. From the gap graph, the existence of the mutation can be demonstrated, but the data lack the power to fully resolve the exact sequence of the gap. Nevertheless, it can be identified and targeted for further experimental study.

5. Overview of the format

Here we provide an informal overview of the format, deferring details to the appendices. The core unit of the format is a FASTA-like construction (§14), that can exhibit a DNA sequence and its connection to other sequences that comprise a sequence digraph (see next section). For example:

```
>x:y;
ACGGACAT
```

Here `x` represents both a DNA sequence and an edge in a graph. This edge is in turn followed by edge y – i.e. there exists an adjacency from edge x to edge y. To facilitate the representation of inversions, the format also allows for adjacencies between forward and reverse complement (RC) of edges – see the next section for more details. Next, the format allows for the direct embedding of an acyclic graph inside FASTA (for an example see §8), and for several convenient specializations of this, in particular 'ambiguities' involving a list of alternatives, tandem repeats, and heterozygous inversions, and cases where the orientation of a sequence is unknown, *e.g.* if it was placed using a genetic map (§15). Finally the format can represent gaps of prescribed distance (§15), with error bars, and gaps that are stuffed with an unresolved graph (for an example see §4).

6. Semantics

The construction of FASTG depends on a core object that we refer to as a 'sequence digraph'. First we provide a mathematical description, then we explain how it is encoded in FASTG. The main challenge is with the encoding of reverse complements.

Mathematical description of sequence digraph.

- A *sequence digraph* consists of a collection of DNA sequences, say $e_1, \ldots, e_n$, also called *edges*, and a collection of pairs xy, called *adjacencies*, where x and y are either edges or their reverse complements, denoted by the addition of a prime character. Very explicitly, this is:

adjacency	interpretation
e_ie_j	follow e_i then follow e_j
e_ie_j'	follow e_i then follow the reverse complement of e_j
$e_i'e_j$	follow the reverse complement of e_i then follow e_j
$e_i'e_j'$	follow the reverse complement of e_i then follow the reverse complement of e_j.

  We note that in a sequence digraph, $e_j'e_i'$ is *not* treated in the same way as e_ie_j, as we will explain shortly.
- Note also that strictly speaking, a sequence digraph is not a graph at all, as we have not specified a notion of vertex. However in many cases one can without ambiguity define vertices and thereby associate a *bona fide* digraph, and we do so frequently in this document to illustrate concepts.
- Now given a sequence digraph having edges $e_1, \ldots, e_n$, a *path* is a word in the alphabet $\{e_1, \ldots, e_n, e_1', \ldots, e_n'\}$ such that each pair of successive symbols are adjacencies.
- To every path we associate a DNA sequence by concatenating the associated edges, or their reverse complements, if so specified. To circular paths we associate circular DNA sequences.

FASTG encoding of sequence digraph.

- The header line for a FASTG record encodes an edge, together with a set of adjacencies, and properties, as follows:
  >Edge:Neighbors:Properties;
  where:

- Edge is the name given to this edge/sequence.
- Neighbors is a list of edges (or their reverse complements, indicated by a ') that follow this edge or the reverse complement of this edge (indicated by a preceding ~).
- Properties is a list of optional properties associated with this edge (discussed later in this document).

The empty Neighbor list (given by ::) is valid, but is only necessary if the Property field is populated.

- Some examples:

>Edge1;	specifies Edge1 but no adjacencies or properties.
>Edge1:Edge2,Edge3;	Edge1 is followed by Edge2 and Edge3.
>Edge1:Edge2,Edge3';	Edge1 is followed by Edge2 and the RC of Edge3.
>Edge1:Edge2,Edge3':prop1=value1;	As above, but with property prop1 defined.
>Edge1:Edge2,Edge3',~Edge4:prop1=value1;	As above, but the RC of edge1 is followed by Edge4.

Properties may be associated with each adjacency using the following notation:

>Edge1:Edge2[prop1=value1],Edge3;	The adjacency between Edge1 and Edge2 has property prop1 defined.

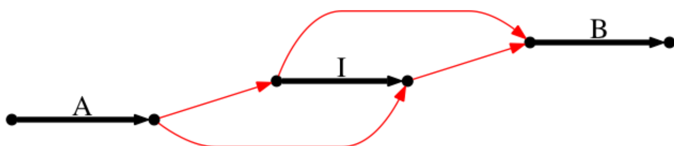
7. Nested graph features

The FASTG format supports global and local graph features. Further nesting of local graph features is allowed only in the case of the `[digraph:…]` construct – see the example in Section 3. Only one additional level is allowed – nesting cannot continue indefinitely. Nesting in this fashion is not allowed for any other construct. In summary:

`digraph` may contain `alt`, `tandem`, `digraph`, `gap`, or `stuffed_gap`.
`digraph` may be nested only once.
`alt`, `tandem`, `gap`, `stuffed_gap` cannot contain additional nested graph features.

8. Examples where reverse complementation is unavoidable

Here we illustrate situations where reverse complement edges (see §6) are needed. Consider first the case of an allelic heterozygous inversion. In fact FASTG provides directly for this, inside a single FASTG record, using the `[digraph:bioriented|…]` notation, and this is the preferred usage because it is better to stay within a single record, which has a single coordinate system. However, to illustrate the digraph notation, we exhibit a direct encoding. Suppose we wish to allow for A{I or I'}B, which we might describe informally using the following picture, in which the four red edges represent adjacencies:



Then we have three edges A, I and B and four adjacencies AI, AI', IB, I'B. In FASTG we would write:

```
>A:I,I';                    encodes AI and AI'
(Sequence of A)
```

```
>I:B,~B;                    encodes IB and I'B
(Sequence of I)
>B;                         encodes no adjacencies
(Sequence of B)
```

Now consider an alternate and 'looser' semantics for an inversion event, in which we know *less*. Specifically, we could add the 'reverse complement' of all adjacencies, thus allowing not just for the 'standard' paths AIB and AI'B, but also for paths like AIA'. To do so we would write:

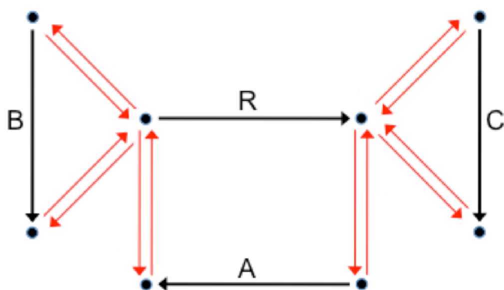
```
>A:I,I';                    encodes AI and AI'
(Sequence of A)
>I:A',B,~A',~B;             encodes IA', IB, I'A' and I'B
(Sequence of I)
>B:~I,~I';                  encodes B'I and B'I'
(Sequence of B)
```

We give one more example. Consider, as in §1, a genome having two circular chromosomes, and three copies of a long repeat R, one on one chromosome and two on the other. However now suppose that the latter two copies are in inverse orientation. Let's use the following shorthand for the genome:

```
chr1:  -- A -- R --
chr2:  -- B -- R -- C -- R' --
```

where R' denotes the reverse complement of R. Now if we have a data set for this genome that includes links long enough to bridge R, then there is no problem: the genome may (in principle) be fully resolved as two circles. But suppose we do not have these long links. Then it is impossible to fully resolve the genome structure from the data: there is more than one solution.

To encode this in FASTG, all we have to do is read off the adjacencies from chr1 and chr2, allowing for circularity:
chr1: AR, RA
chr2: BR, RC, CR', R'B
and then add in the reverse complements of these adjacencies:
chr1: R'A', A'R'
chr2: R'B', C'R', RC', B'R.
This is very much in the spirit of the 'loose' interpretation of a heterozygous inversion described above, and informally illustrated as follows, with red arrows representing the twelve adjacencies:



It is easy to encode this assembly in FASTG, as all we have to do is copy the twelve adjacencies into the notation:

```
>A:R,~R';                encodes AR and A'R'
(sequence of A)
>B:R,~R;                 encodes BR and B'R
(sequence of B)
>C:R',~R';               encodes CR' and C'R'
(sequence of C)
>R:A,C,C',~A',~B,~B';    encodes RA, RC, RC', R'A', R'B and R'B'.
(sequence of R)
```

To illustrate how this works, here are the paths through the graph that reconstruct the true genome:

```
chr1:  A R A
chr2:  B R C R' B
```

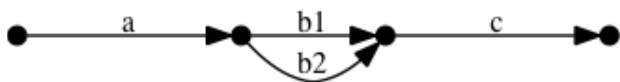
There are other paths through the graph that reconstruct 'fake' genomes. The data do not allow us to distinguish between the true genomes and these fakes.

9. More toy examples of assemblies

(a) This example exhibits an acyclic graph, embedded in an assembly.

```
>xxx;
GTAAAAACTAC
ATATATGTTTTT [12:digraph:path=(a,b1,c), start=a, end=c |
              >a:b1,b2;
              ATATAT
              >b1:c;
              G
              >b2:c;
              C
              >c;
              TTTTT               ]
ACACACAC
```

This embedded acyclic graph can be visualized as follows:



There are two paths through the graph: a→b1→c, corresponding to the sequence ATATATGTTTTT, and a→b2→c, corresponding to the sequence ATATATCTTTTT. The semantics guarantee that one or more of these sequences is present in the sample (at the given locus).

The notation `a,b1,c` declares a canonical path through the graph, and from this one deduces the canonical FASTA representation for the assembly

```
>xxx;
```

```
GTAAAAACTAC ATATAT G TTTTT ACACACAC
```

(b) Example (a) has the following equivalent and much simpler representation:

```
>xxx;
GTAAAAACTAC
ATATAT G[1:alt|G,C] TTTTT
ACACACAC
```

10. Copy number statistics

Here we describe a generate framework for quantifying information about copy number, and provide certain simple abbreviations that encapsulate the most common cases. We expect that in the vast majority of situations, only these common cases would be used.

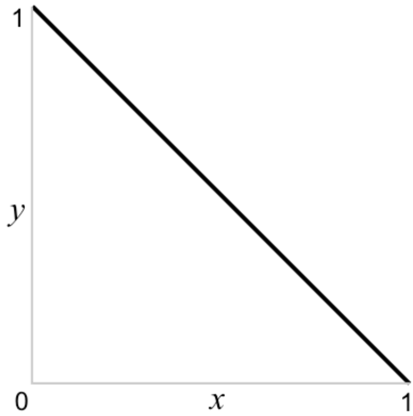
We provide three vehicles for representing copy number information. First, as described earlier, the `tandem` notation provides for specification of the number of copies of a tandem repeat.

Second, we allow for representation of 'external' copy number. This provides information about the relative abundance of the connected components of the top-level graph in a FASTG file. It is encoded via the property `cn_external`, that may be assigned to any top-level FASTG record (but only consistently: records in the same component may not be assigned different values). The values for `cn_external` are to be interpreted relative to each other. For example, suppose an assembly consists of mitochondrial and nuclear genomes, present as single FASTG records and of molarity 10:1 in the sample. Then this information might be represented as

```
>mitochondrial_genome::cn_external=10;
...
>nuclear_genome::cn_external=1;
...
```

We leave specification of error bars for `cn_external` to a future version of FASTG, should they prove needed.

Third, suppose we have alternative sequences *x*, *y*, for example as in `[1:alt|A,T]`. We may be able to express information about their likelihood of presence in the sample and their relative abundance. This could be encoded in a probability density function (pdf) over the diagonal segment connecting the points (0,1) and (1,0),



with each point on the segment representing a hypothetical distribution of 'copy number', summing to one. In this version of FASTG we provide for the specification of the pdf in the case where it is discrete, *i.e.* concentrated at a finite set of points, and in the continuous case where there is no information, *i.e.* equal probability of all cases. We leave open the possibility that future versions of FASTG would allow specification of other continuous pdfs, as needed.

Such discrete pdfs may be expressed via a property `cn_discrete_pdf` (where cn is short for copy number), as the following examples illustrate:

(a) semantics: we are certain that two alleles are present in equal proportions
  pdf: concentrated at the single point (0.5,0.5)
  long form: `cn_discrete_pdf = ( ((0.5,0.5),1) )`
  short form: `allele`
  example: `[1:alt:allele|A,T]`

(b) semantics: we are certain that only one allele is present, and assign equal probabilities to both
  pdf: concentrated equally at two points (1,0) and (0,1)
  long form: `cn_discrete_pdf = ( ((1,0),0.5), ((0,1),0.5) )`
  short form: `exclusive`
  example: `[1:alt:exclusive|A,T]`

(c) semantics: we are certain that only one allele is present, and assign probability 0.99 to the first
  pdf: concentrated at the two points (1,0) and (0,1), with 99% of the mass on the first point
  long form: `cn_discrete_pdf = ( ((1,0),0.99), ((0,1),0.01) )`
  short form: `exclusive=(0.99,0.01)`
  example: `[1:alt:exclusive=(0.99,0.01)|A,T]`

(d) semantics: we have no information about the relative presence of the two alleles
  pdf: uniform along the line segment
  long form: none
  short form: (default)
  example: `[1:alt|A,T]`

These concepts and notations generalize readily to the case of three or more alternatives.

Note that to avoid limitations of floating point arithmetic, we allow for scaling of numbers in the discrete pdf representations, for example

```
cn_discrete_pdf = ( ((1,1,1),1) )
```

corresponding to three alleles present in equal proportions, understanding that this corresponds to the pdf being concentrated at the single point (1/3,1/3,1/3).

The same representation of copy number for alternatives is allowed for the `digraph` notation, provided that represents a collection of parallel edges (like `alt` does).

11. Assembly assessment

(a) Investigators will want to report the properties of assemblies and we note that replacement of the traditional format by a new format will entail changes to that reporting. While falling outside the format *per se*, this has some bearing on the language used by it. We also note the potential value in introducing a standard for assessment. Since a mechanism for canonical conversion to conventional FASTA is provided, in the short term reporting can continue to use current methods applied to the canonical FASTA, although this does not take full advantage of the new format.

(b) Assemblers should be penalized for all ambiguity that is not present in the genome. However, it is clearly better to report an ambiguity than to make an error. It is not clear exactly how ambiguity should be counted.

(c) The terms contig and scaffold need to be clarified, so that (for example) one could report the N50 contig or scaffold size. For example, we would argue that stuffed gaps should be *between* contigs (rather than within them), for the following reasons:
   - Inclusion within contigs could result in 'huge' contig sizes, causing the community to think that assembly is a 'solved' problem.
   - It makes sense to incentivize the development of data and algorithms that will lead to the full resolution of gaps.

   At the same time, it makes sense to also incentivize the 'stuffing' of gaps, so this should be reflected somehow in assembly assessment statistics.

(d) In the case of a controlled experiment, where a reference sequence for the genome is available, we note that the assembly is regarded as correct if it traverses a valid path, *i.e.* through at least one alternative (in the case of ambiguities), and similarly for the case of stuffed gaps.

12. Miscellaneous issues

- While users may want to see the evidence for an assembly, typically exhibited by read alignments, we recommend that these data be presented separately from the assembly *per se*. One reason for this is that the evidence data could easily be 100 times larger than the assembly. However the format might specify the mechanism for presentation of evidence. For example Binary Alignment/Map (BAM) might be used, but there would need to be a naming convention to coordinate between assembly and evidence files. Also the BAM format might need to be adopted to insure that it is robust enough to handle reads that span across branch boundaries.

13. Appendix: character set and names

Here we begin the detailed formal description of the assembly notation. First, the alphabet for the format consists of :

`a-ZA-Z0-9_.,[]|=>:;-~()#”`

and white space (defined here to be blanks, newlines and tabs), which ignored, except between matched pairs of double quotes (thus allowing for comments and other literal strings). White space may be inserted to enhance readability.

Here are a few definitions:

- **name_string** := {`a-zA-Z0-9_`}+

- **value_string** := {`a-zA-Z0-9_().,” `}+

subject to:

- For any comma in the string, the number of left and right parentheses to the left of it may not be equal. This restriction is needed to allow unambiguous parsing of FASTG.
- The double quote character may only be used in a ‘literal’ **value_string**, having the form ”...”, where the intervening text is a string of length zero or more from the **value_string** character set, but without double quotes. Note in particular that blanks are allowed in such a literal string, and treated as part of the value, whereas white space is ignored everywhere else in FASTG. Note also that the quotes themselves are not treated as part of the value, so that `x=1` and `x=”1”` have the same semantics, but that
  `x=”Homo sapiens”` has no quote-free equivalent.

- **property_string** := a comma-separated list consisting one or more strings of the form

- *name*=*value*, where *name* is a **name_string** and *value* is a **value_string**

or

- *name*, where *name* is a **name_string**, which is short for the ‘flag’ *name*=1.

The FASTG format defines a small number of properties. Moreover, we allow the arbitrary introduction of user-defined properties, with the intention that these be gradually and selectively absorbed into the format if they turn out to have general utility.

Each FASTG file starts with the following special text, with the appropriate version number substituted in:

`#FASTG:begin;`
`#FASTG:version=*.*;`
`#FASTG:`*properties*`;`

where *properties* is a **property_string**, defining global properties of the FASTG file. It is possible to combine these into a single line:

`#FASTG:begin:version=*.*:`*properties*`;`

The list of properties can be split over multiple lines, each starting with `#FASTG`, but individual properties shouldn’t be split:

`#FASTG:`*properties*`;`

`#FASTG:`*more_properties*`;`

Each FASTG file ends with the following special text:

`#FASTG:end;`

On `#FASTG` lines spaces are ignored (as elsewhere, except in comments and quoted strings, as described below). The semicolon on the `#FASTG:end;` line is regarded as the terminating character for the FASTG file.

The following global properties are defined in this version of FASTG:

```
organism
assembly_program_name
assembly_program_version
assembly_run_date
assembly_run_number
assembly_name
comment
```

All text occurring between a `#` and a newline is treated as a comment and ignored unless it begins with `#FASTG`.

Every sequence in the format is assigned a name. These are relative, so if name x_2 appears within something named x_1, then its 'absolute' designation would be $x_1 : x_2$.

14. Appendix: number notation

■ **nonnegative_integer_string** := `0 | {1-9}{0-9}*`

■ **integer_string** := `0 | {1-9}{0-9}* | -{1-9}{0-9}*`

■ **nonnegative_integer_range_string** := *n* or *m*`..`*n*
where:
- *m*, and *n* are **nonnegative_integer_strings,** and $m < n$

■ **integer_range_string** := *n* or *m*`..`*n*
where:
- *m*, and *n* are **integer_strings,** and $m < n$

■ **abbreviated_nonnegative_integer_list_string** := $x_1, \ldots, x_n$
where:
- each x_i is a **nonnegative_integer_range_string**. Note that duplicates in the list are to be ignored.

Example: `15,10..20` is a way of listing the integers from 10 to 20, with 15 first.

■ **abbreviated_integer_list_string** := $x_1, \ldots, x_n$
where:

each x_i is a **integer_range_string**. Duplicates in the list are to be ignored.

■ **positive_float_string** :=
{1-9}{0-9}+.{0-9}+ | 0.{0-9}+ | {1-9}{0-9}*

■ **float_string** := 0 or x or $-x$ where x is a **positive_float_string**.

■ **probability_string** := a **float_string** p, with $0 < p \leq 1$.

15. Appendix: notation for bases and sequence digraphs

■ **base_string** := {ACGT}*

We deliberately exclude N from the **base_string** definition, because historically, N has been 'overloaded': it has been used to define gaps (as part of a run), ambiguous bases (meaning A or C or G or T), and also to mark other types of uncertainty, including possible indel errors in an assembly. We prefer to require that assemblers precisely specify their intended meaning, which should be possible using the [gap...] and [alt...] constructs of this format. Note that empty **base_strings** are allowed. Note also that IUPAC ambiguous bases and lower-case bases are not allowed.

■ **canonical_base_string** := {ACGTN}*

We allow N in the **canonical_base_string**, to support the encoding of gaps. These strings can only occur if they precede a 'bracket' expressions (**embedded_diagraph_string**, **ambiguity_string**, **tandem_string**, **gap_object_string**, and **stuffed_gap_object_string**), they cannot occur alone. Note that empty **canonical_base_strings** are allowed. Note also that IUPAC ambiguous bases and lower-case bases are not allowed.

■ **fasta_string** :=
>*edge* **:** $edge_1$,...,$edge_n$ **:** *properties* **;** *bases*
or
>*edge* **:** $edge_1$,..., $edge_n$ **;** *bases*
or
>*edge* **;** *bases*
where:
- *edge*, $edge_i$ are **name_strings**
- *bases* is a **base_string**
- *properties* is a **property_string**. Currently the only recognized property in this context is cn_external (§8).

It is conventional (but not required) to have a newline after the semicolon. The semicolons are needed to allow for consistent handling of white space.

■ **digraph_string** := $b_1,...,b_n$
where:
- each b_i is a **fasta_string**, or a **fasta_string** followed by a ' to indicate the RC. The order of the b_i does not matter.

The semantics are as follows. A *path* through a digraph of this type hops along a sequence of edges, corresponding to either **fasta_strings** , referred to here as sequence edges. Normally all edges must be read forward, unless indicated by a following ' which indicates the RC of an edge. Only paths that follow valid adjacencies between the edge are allowed.

16. Appendix: basic FASTA strings

A **basic_fasta_string** is defined to be either a **base_string**, or any of the other strings defined in this section.

In this section we define the five 'bracket' expressions plus their canonical representation, which all have the form

canonical_base_string [*size* **:** *type* < **:** *properties*>...]

where *size* is the length in bases of the preceding canonical FASTA sequence, *type* is one of `digraph`, `alt`, `tandem` or `gap`. The properties may always include `name=`*name*, where *name* is a **name_string**, thereby assigning a name to the bracket expression. For purposes of FASTG processing, if a bracket expression is not explicitly assigned a name, it will be assigned a unique name by the processor.

■ **embedded_digraph_string** := [*size* **:** `digraph` < **:** *properties*> | *g*]
where:
- *size* is a **nonnegative_integer_string** giving the size of preceding canonical FASTA sequence.
- *properties* is a property_string; apart from `name` and `cn_discrete_pdf` (§8), only the following are assigned semantics at present: `unoriented` and `bioriented`, and presence of both is not allowed; also we require
  `path=`(s_1**,** ... **,** s_k)
  where each s_i is a **name_string**, representing edge names in *g*, defining a canonical path through the graph
- two properties, `begin` and `end`, are used to indicate the entry and exit edges to and from the graph respectively. They may be omitted only if the graph is simply a set of parallel alternative edges.
- *g* is a **super_digraph_string**, representing a digraph.

Notes:
- This definition is recursive: we do not define **super_digraph_strings** until the next section.
- Cycles are not allowed within an embedded digraph, as it would allow for unbounded cycling. Note that it is acceptable to include cycles within a stuffed gap (defined below), since in that context the gap size defines a limit on the number of times the cycle would be traversed.
- If the `unoriented` flag is present, then the construct represents either the given sequence, or its reverse complement, implying that at least one is present at the given locus. Such data arise naturally from genetic mapping, and also from long inverted repeats that are not bridged by read pairs. Note that in such situations it is probably not possible to rule out the possibility that both the sequence and its reverse complement are present (*cf.* below).
- If the `bioriented` flag is present, then both the sequence and its reverse complement are present, as would be the case for an allelic heterozygous inversion.

■ **ambiguity_string** := [*size* **:** `alt` < **:** *properties*> | x_1**,** ... **,** x_n]
where:

- *size* is a **nonnegative_integer_string** giving the size of preceding canonical FASTA sequence.
- *properties* is a **property_string**
- each x_i is either a **base_string** or else of the form y_i **:** *properties* where y_i is a **base_string** and *properties* is a **property_string.**

Semantics: it is 'guaranteed' that at least one of the x_i appears at the given locus.

Note that it matters which x_i is first, because that x_i is inserted in the canonical FASTA representation. Otherwise, order doesn't matter.

Note that an **ambiguity_string** is a special case of an **embedded_digraph_string**.

■ **tandem_string** := `[`*size*`:tandem` <`:`*properties*>`|`*bases*`]`
where:
- *size* is a **nonnegative_integer_string** giving the size of preceding canonical FASTA sequence.
- *name* is a **name_string**
- *properties* is a **property_string**, with a single recognized and required property
  `size=(`*l*`)`
  where *l* is an **abbreviated_nonnegative_integer_list_string**
- *bases* is a nonempty **base_string.**

Note that a **tandem_string** is a special case of an **ambiguity_string.** The same comments about order apply.

■ **gap_object_string** := `[`*size*`:gap` <`:`*properties*>`]`
where:
- *size* is a **nonnegative_integer_string** giving the size of preceding canonical FASTA sequence.
- *properties* is a **property_string**, with (apart from `name`) a single recognized property
  `size=(`*l*`)`
  which is required, where *l* is an **abbreviated_nonnegative_integer_list_string**.

The canonical FASTA representation consists of max(1, l_1) `N`s.

■ **stuffed_gap_object_string** := `[`*size*`:gap` <`:`*properties*>`|`*g*`]`
where:
- *size* is a **nonnegative_integer_string** giving the size of preceding canonical FASTA sequence.
- *properties* and *l* are as in a **gap_object_string**; the properties may include
  `path=(`s_1`,…,`s_k`)`
  where each s_i is a **name_string**, representing edge names in *g*, defining a canonical path through the graph
- *g* is a **digraph_string**
- two properties, `begin` and `end`, are used to indicate the entry and exit edges to and from the graph respectively. They may be omitted only if the graph is simply a set of parallel alternative edges.
- if `path` is unspecified, the canonical FASTA representation is as in a **gap_object_string.**

Note that here we allow the digraph to be cyclic, however it is bounded by the constraint defined by *l*.

Semantics: it is guaranteed that at least one path from a `begin` edge to an `end` edge (and within the given size bounds) is present in the sample at the given locus.

17. Appendix: assemblies

- **super_fasta_string** := a **fasta_string**, except that instead of *bases* being a **base_string**, it is a concatenation of one or more **basic_fasta_strings**.

- **super_digraph_string** := a concatenation of zero or more **super_fasta_strings**

By definition, an *assembly* is a **super_digraph_string**.

18. Appendix: required software

Certain software would need to be supplied with FASTG:

(1) Validator. This would determine if a given FASTG file is valid.

(2) Flattener. This would take as input a FASTG record (or a snippet from such a record, such as a bracket expression), and produce as output

- a file providing the canonical FASTA for the FASTG
- optionally, a file of markups of the FASTA
-

Both of these programs would need to be highly portable, to trivialize installation. For example, both could be self-contained perl scripts.

Both programs would be versioned in accordance with the version of FASTG, together with minor versions corresponding to improvements within a given FASTG version.

19. Appendix: FASTG to FASTA conversion

The FASTG format has been designed to make the conversion to plain FASTA as simple and easy as possible. Because the canonical FASTA sequence is given just prior to the FASTG construct that describes it in more detail, it is possible to convert a FASTG file simply by stripping out the FASTG constructs. These constructs are given between square brackets "[]".

For example:

```
>MyFirstContig;
GTAAAAACTACATATAT G[1:alt|G,C] TTTTTACACACAC
```

Becomes

```
>MyFirstContig;
GTAAAAACTACATATAT G TTTTTACACACAC
```

Where the spaces have been added for clarity, the canonical sequence is in red and the FASTG construct it is derived from is in **bold**.

This has the added advantage that the co-ordinate system between the source FASTG and the derived FASTA are consistent.

20. Appendix: FASTG markup

FASTG is its own markup.

After FASTG to FASTA conversion, it would be useful to retain the FASTG constructs in a separate file that could be cross referenced with the FASTA. Rather than define yet another markup format, the FASTG constructs are reused, but with the addition of a base offset. For the example from Section 18, the markup would simply be:

```
>MyFirstContig;
17 [1:alt|GC]
```

Where the additional base offset is indicated in **bold**. The combination of this offset and the canonical sequence size (the 2nd field) completely identifies the location of the FASTG feature in the FASTA file. It is therefore also possible to take a FASTA file, plus the FASTG markup and regenerate the original FASTG file.
Since the **fasta_string** is reproduced exactly in the 'markup', the global graph structure and properties are also retained.

The advantages of this markup system are its simplicity and easy of construction. A very basic tool could take a FASTG file and generate markup and FASTA, and vice versa. However, it is recognize that in some situations this markup format will be insufficient. In particular, the recursive nature of the FASTG format remains locked up in the markup, and has not been unrolled. It is imagined that other markup schemes will be proposed in the future to address this and other shortcomings, in addition to the development of tools to covert FASTG markup to existing annotation schemes.

21. References

[1] Pearson WR, Lipman DJ. Improved tools for biological sequence comparison. *Proc Natl Acad Sci USA* **85** (1988), 2444-2448.

[2] Pearson WR. Rapid and sensitive sequence comparison with FASTP and FASTA. *Methods Enzymol* **183** (1990), 63-98.

David B. Jaffe, Iain MacCallum, Daniel S. Rokhsar, Michael C. Schatz