

A CGL Tutorial

Abstract

Background

Sequence alignment algorithms have revealed much about evolution at the nucleotide and amino-acid level, but little is yet known about the structural evolution of genes; that is to say, how their intron-exon structures, alternative splicing, and UTRs change overtime. Genome annotations comprise an invaluable resource for such studies because they describe the essential parts of a gene and their relationships to one another. In order to facilitate the use of genome annotations for comparative genomics we have developed an open-source software library called CGL.

Results

The primary strengths of CGL are that it allows one to relate the parts of one annotation, e.g. its transcripts, exons, introns, UTRs, etc to those of another annotation by means of a sequence alignment. Thus, CGL greatly simplifies the task of writing scripts to compare genome annotations to one another. Here, we describe the software, explain how to use it.

Conclusions

CGL makes it possible to study evolution from the perspective of gene structure and will likely provide an important adjunct to more traditional approaches that are based solely upon protein and nucleotide similarities. CGL provides an easy means to explore, compare, and characterize genome annotations, allowing users to ask and answer significant biological questions about genome-annotations without becoming bogged down in tangential programming issues.

Background

Sequence alignment algorithms have revealed much about evolution at the nucleotide and amino-acid level, but little is yet known about the structural evolution of genes; that is to say, how their intron-exon structures, alternative splicing, and UTRs change overtime. Genome annotations comprise an invaluable resource for such studies because they describe the essential parts of a gene and their relationships to one another.

Despite their potential utility for comparative studies, however, annotations remain primarily something 'seen' in a genome browser [1], rather than something used for computation, and many bio-informatics practitioners continue to think of annotations as little more than sources of transcript and protein FASTA-files. This is an unfortunate state of affairs, as every sequence alignment implicitly aligns two gene-structures (FIGURE 1), and exploring the interplay between the evolution of gene-structure and sequence-similarity is a promising new area for bio-informatics made possible by genome-annotations. To date, however, work in this area has been limited due to the absence of publicly available software tools for computing on annotations. Here we present an easy to use, open-source software library designed to facilitate the use of genome annotations as substrates for computation; we call it 'CGL' an acronym for *Comparative Genomics Library*, and pronounce it 'Seagull'.

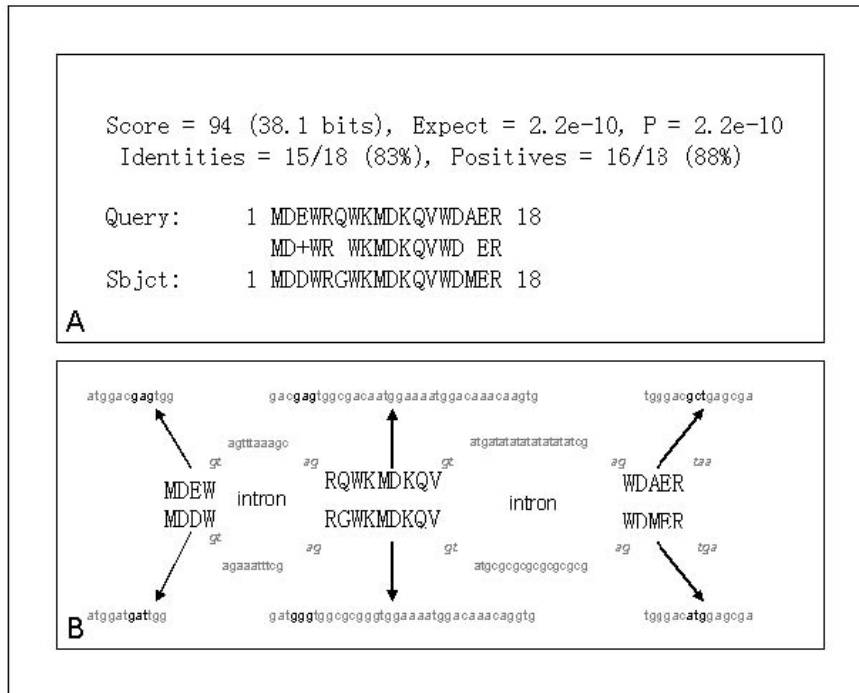


Figure 1. Every sequence alignment implicitly aligns two genes. Panel A shows a typical BLASTP HSP giving the alignment of two proteins; **panel B** outlines some of the relationships between those two protein's corresponding genome annotations that are *implied* by the protein alignment. Aligning two proteins, for example, also implicitly aligns the transcripts that produced them allowing one to ask questions about changes in codon usage. Other implied relationships include introns: knowledge of where their splice junctions fall relative to the protein allows one ask how often the introns of genes with similar proteins occur in similar places; likewise, knowledge about where the splice junctions on a transcript fall relative the protein it produces, can imply can alignment between the annotated exons as well. CGL is a software library designed to ask and answer questions such as these.

CGL has several features that greatly facilitate comparative genomics using genome-annotations. First CGL can convert annotations in GenBank, Ensembl and FlyBase format into a single standardized graph-based description that conforms to the Sequence Ontology [2] and GMOD database schema [3]. Because these documents completely describe each annotation in a formal, logical sense, they enable software that can easily move from a position on any part of a genome annotation to the equivalent position on some other part. For example, given the position of an intron-exon junction on genomic sequence, the relative position of that junction on an annotated protein can be recovered. We term such coordinate transforms 'metaPositions', and detail their use below. A second key feature of CGL is that it extends the BioPerl Blast Hit and HSP classes [4], in such a way that the coordinate of any aligned residue can be extracted from a BLAST alignment. The equivalent position of that residue's partner in the alignment can also be recovered.

Taken together these two functionalities allow CGL users to exploit genome annotations and sequence alignments in new ways. In point of fact, CGL makes it possible to 'blast annotations' rather than sequences against one-another, making it possible to infer a wide array of information about conservation of gene structure from a BLAST report, even from an un-annotated target genome, and even in the absence an assembly.

A standardized machine-readable description of an annotation

Normally one thinks of genome annotations in the context of large web-based warehouses such as GenBank [5], Ensembl [6], SGD [7], FlyBase [8], or WormBase [9]. These databases all employ robust data-models for describing annotations and distribute these data in a variety of markup languages such as GFF and XML; other repositories also exist in both academia and industry, each with their own distinctive data-models. For better or worse, however, there exists no central repository where one can go to get every genome-annotation represented using a single data-model, and in a single markup language; nor is there likely to be one so long as genome sequencing and annotation remain a decentralized, community activity.

This means that a researcher interested in one or a few genome-annotations from a variety of organisms is often faced with incompatible formats for each annotation. In practice then, using genome annotations for comparative genomics often entails writing complex software to convert these disparate formats in to a single usable data-representation.

CGL circumvents these difficulties, and shields users from the complexities that surround data-conversion and representation, as the library comes with a script, `cx-genbank2chaos.pl` (<http://www.fruitfly.org/chaos-xml/bin/>), that allows users to convert GenBank, Ensembl, and FlyBase annotations into a single file format that we term Chaos.xml. These XML documents are graph-based descriptions of annotations that make use of the Sequence Ontology [2] to describe the data they contain.

Using these files as inputs to CGL has two chief advantages. First, they minimize overhead, especially for researchers who are interested in only a few annotations and have neither need nor desire to install local copies of the various annotation databases. Probably, the biggest advantage Chaos.xml documents confer, however, is that they greatly facilitate the logical partitioning of annotations into their explicit and implicit parts, and explorations of the relationships among these parts.

Explicit & Implicit Parts and Coordinate Relationships

Every genome annotation contains both explicit and implicit parts. There also exist coordinate relationships among these parts. Explicitly annotating a transcript's exons on the genome, for example, also implicitly annotates introns. Likewise, annotating a start codon on that transcript creates an implied coordinate relationship between the ATG on the transcript, and the position of that ATG on the genome. The volume of explicit and implicit information contained in an annotation is therefore very large. Thus, those parties creating and distributing annotation information must make practical decisions as to which parts and relationships they explicitly describe in their data models. Clearly, not every part or relationship is of equal interest to the community as a whole: most researchers are more interested in exon coordinates on the genome than they are the phases of those exon junctions relative to a protein. Thus most annotation databases attempt to meet the needs of the 'average' researcher by distributing flat-files that explicitly describe an annotation's most frequently 'used' parts and relationships; and by providing frequently-used logical partitions of annotation data, such as fasta-files of transcripts, and proteins. While these data meet most researchers' needs, they often are of little help for those interested in digging deeper into the *in silico* structure of a gene.

Distributing gargantuan flat-files that exhaustively enumerate every conceivable relationship between the parts of an annotation is simply not a workable solution. And providing pre-written databases queries granting researcher access to them all is task best left to Sisyphus; on the other hand, it's precisely these sorts of data and relationships that CGL is designed to grant its users easy access to.

Access to the Explicit and Implicit Parts of a gene-annotation

CGL is written in Object Orientated PERL and provides simple and intuitive access to an annotation, its parts both explicit and implicit, and the coordinate relationships that obtain between them. Note that CGL comes with a variety of 'starter' scripts and sample data in its `script` and `sample_data` sub-directories. Working examples of most of the manipulations described below can be found in `script/cgl_tutorial` and `script/cgl_phat_tutorial`.

CGL's base object is an 'Annotation'. A Chaos XML file is converted in to an Annotation object with the following call:

```
$annotation = new CGL::Annotation($chaos_file).
```

Though a Chaos XML file can contain several gene annotations, analysis of an annotation usually begins at the individual gene level. CGL provides several methods for extracting a gene object from an annotation object. Probably the most straightforward approach is to access a gene by its id:

```
$g = $annotation->get_gene_by_id('id').
```

Next fetch the genomic contig; this is the DNA backbone on which the gene of interest resides:

```
$c = $annotation->contig(0).
```

To navigate within a gene, all programmers need keep in mind is a simple model of what of a gene is, and what its parts consist of. Formally, CGL's gene model is that of the Sequence Ontology [2], so the model is explicit and rigorous, but it's also simple. CGL considers a gene to consist of one or more transcripts; these transcripts are composed of one or more exons. To fetch the first annotated transcript associated with a gene use the call shown below; this is the transcript with the 5'-most annotated transcription start-site:

```
$t = $g->transcript(0).
```

If that gene has more than one annotated transcript, they can be accessed in order of their transcription start-sites from 5-prime to 3-prime by incrementing the argument to the method, e.g., 1, 2, and so on. The transcript method will return 'undef' if no such transcript object exists. This is true of every CGL 'Feature' method. *Attempts to access a non-existent part or coordinate relationship always return undefined.* This feature of the CGL allows users to easily ask if a part or relationship exists before proceeding with further analyses, i.e.,

```
next unless defined($g->transcript(2)).
```

To access the protein associated with a transcript use the call shown below. If that transcript has more than one annotated translation, those translations can be accessed by incrementing the argument to the method, just as was described above for the transcript method. Again, if the i^{th} translation doesn't exist, or your gene is a non-coding RNA gene, the method will return undefined.

```
$p = $t->translation(0).
```

The exons of a transcript are accessed in the same manner:

```
$e = $t->exon(1).
```

This call will return the 2nd exon of the transcript, or 'undef' if the transcript contains only a single exon. Note that if a gene has two alternatively spliced transcripts that share an exon in common, then both transcript objects share that exon object in common. This is both logical and in keeping with the specifications of the Sequence Ontology [2]. It also greatly facilitates many analyses. Shared exons—and shared parts in general—can easily be discovered using the id() method, as these are always unique within the context of a given CGL::Annotation, e.g.,

```
$e->id().
```

In every case the sequence associated with an object can be accessed through the residues method:

```
$p->residues().
```

The above call above will return the amino acid sequence of that protein object; likewise calling the same method on a transcript, exon, or contig object will return the appropriate nucleotide sequence.

Introns provide an example of an implicit part of an annotation. Though not explicitly described in most annotation data-models, including the Chaos data-model, introns are of interest to a large enough number of researchers to merit their own predefined class in CGL. Like exons, introns are associated with transcripts. To access an intron use the call

```
$i = $t->intron(0).
```

The behavior of this method is identical to the exon method, and like that method, if two transcripts of a gene contain the same pair of exons, then those two transcripts will also share the same intron. Hence, an intron object may be shared in common with one or more transcript objects. Like the explicit parts of a CGL annotation, the sequence of an intron is accessed through the 'residues' method.

Access to Explicit and Implicit Coordinate Relationships

The metaPos() method allows CGL users to explore explicit and implicit coordinate relationships between the parts of an annotation. Using this method, a programmer can relate any position on any part of an annotation to the equivalent position on any other part of that annotation for which that operation makes semantic sense. It is possible to move, for example, from a position in a protein sequence to the equivalent position in an exon, transcript or contig and *vice versa*. The syntax of the MetaPos() method, is 'reverse polish', and is used as follows:

```
<desired offset on $b> = $a->metaPos($b, <known offset on $a>).
```

The explicitly annotated start of an exon on the genomic contig, for example, can be recovered with the following call:

```
$e->metaPos($c, 0),
```

where \$e, and \$c are exon and contig objects as described in the previous section, and 0 refers to the offset of interest with the object calling the metaPos method. Note that because the Chaos data-model is 'zero space based', an offset of 0 on the exon corresponds to its beginning; likewise the length of the exon corresponds to its end. Thus, the explicitly annotated end of an exon on the genomic contig can be obtained with the call

```
$e->metaPos($c, $e->length()).
```

Notice that the only difference between this call and the one above it involves the second argument to metaPos(). To obtain, for example, the position of the 3rd base of an exon on the genomic contig—an implied coordinate relationship, call the method with that offset, i.e.

```
$e->metaPos($c, 2).
```

The metaPos() method provides uniform access all coordinate relationships among the parts of an annotation, *explicit and implied*. For, example, to recover the start of translation on a transcript use the call

```
$p->metaPos($t, 0).
```

Likewise the call below will return the position of the stop codon on a transcript.

```
$p->metaPos($t, $p->length).
```

The equivalent position on the protein of any position of the transcript can be recovered just as easily:

```
$t->metaPos($p, 341).
```

Note that relationships between an offset on a transcript and the equivalent position on a protein are complicated by the genetic code. Since nucleotides are translated three-at-a-time, there exists a 'many to one' relationship between positions on the transcript and positions on the protein. In order to avoid ambiguity, the metaPos() method returns a float in such cases, the mantissa of which gives the amino acid, the remainder the phase within that amino-acid's codon. For example, a value of 21.33333 would mean that an offset of 341 on the transcript corresponds to amino acid 21 of the protein in phase 1 of the 21st codon. A value of 21.6666 would denote the same amino acid, but in phase 2. If the return value lacks a remainder, it means that the phase is 0. In cases where the offset on the transcript lies in UTR, the call shown above will return 'undef'. This is true of all calls to the metaPos() method: if the requested meta position does not logically obtain, the method will return undefined.

Some coordinate relationships require sequential calls to the metaPos() method. To recover the position of a protein's methionine on the genomic contig, for example, first find its position on the transcript:

```
$start = $p->metaPos($t, 0).
```

Then transpose that position onto the genomic contig:

```
$t->metaPos($c, $start).
```

Multi-step meta-positions such as these can be abbreviated as:

```
$t->metaPos($c, $p->metaPos($t, 0)),
```

and are required whenever the sought after coordinate relation involves a protein and some object that is not a transcript.

The examples discussed so far illustrate two of the principle features of the metaPos() method. First, no matter what parts of the annotation are involved, the grammar and syntax of the call remains the same and has the form:

```
<desired offset on $b> = $a->metaPos($b, <known offset on $a>).
```

Second, it makes no difference whether the sought after coordinate relationship is explicitly described in the Chaos data-model or implicit. Any semantically meaningful use of the metaPos() method will return a value. This means that users of the CGL library are no longer slaves to the details of the annotation data model—any coordinate relationship that they can imagine can be recovered in a single line of code. This frees CGL users to discover explore and explore biologically meaningful relationships amongst the parts of an annotation without even considering whether or not those relationships are explicit or implied.

Manufacturing Implicit Parts for further analyses

Most bioinformatics analyses involve sequence analyses. CGL facilitates these analyses by providing easy access to the sequences that correspond to the explicit and implicit parts of an annotation. Access to the sequence of explicit parts is granted through the residues() method (see § Access to Explicit and Implicit Parts for details). The large number and diversity of implied parts, however, precludes the existence of a single, generic residues() method for the implied parts of an annotation. This means that CGL users interested in the sequence of a particular implied part need to manufacture that sequence themselves. CGL provides the easy means to do so.

UTRs are a good example. These sequences are of interest to many researchers, and yet can be difficult to obtain. To recover the sequence of an annotated transcript's 5- & 3-prime UTRs with CGL, first use the metaPos() method to obtain the start and stop of the translation, where \$t is the 'transcript of interest', and \$p, one of its translations:

```
$start = $p->metaPos($t, 0).  
$stop = $p->metaPos($t, $p->length()).
```

The sequences of the 5 and 3-prime UTRs, respectively, can then be obtained with the following two lines of code:

```
substr($t->residues, 0, $start);  
substr($t->residues, $stop).
```

Or, if the user so desires, the entire procedure can be abbreviated in the form:

```
substr($t->residues, 0, $p->metaPos($t, 0));  
substr($t->residues, $p->metaPos($t, $p->length)).
```

The sequences of more exotic implied parts of an annotation are just as easy to manufacture. The portion of a protein corresponding to a given exon provides an instructive example. To manufacture this sequence, first identify the exon's implied begin and end on the transcript of interest:

```
$e = $t->exon(1);  
$t_begin = $e->metaPos($t, 0);  
$t_end = $e->metaPos($t, $e->length).
```

Next use the metaPos() method to identify the equivalent positions on the protein:

```
$p_begin = $t->metaPos($p, $t_begin);
```

```
$p_end = $t->metaPos($p, $t_end).
```

The desired portion of the protein's amino-acid sequence can then be obtained with a call to the PERL substring function:

```
substr($p->residues, $p_begin, $p_end - $p_begin).
```

Finally, users whose research is centered on a particular implied part of a gene may find it convenient to write their own subroutines using CGL, thus reducing the amount of scripting required to manufacture such sequences still further, e.g.:

```
print utr_sequence($t, $p, 5)."\n";
```

```
sub utr_sequence {  
    ($t, $p, $type) = @_ ; if ($type == 5){  
        return substr($t->residues, 0, $p->metaPos($t, 0));  
    }  
    else {  
        return substr($t->residues, $p->metaPos($t, $p->length));  
    }  
}
```

These examples demonstrate a fundamental design paradigm of CGL: less is more. The utility of CGL springs from its simplicity. Rather than provide a bewildering array of exotic methods for manipulating annotations, CGL provides a few essential methods that can be combined in different ways to manufacture any implicit part of an annotation. Most tasks can be accomplished with recourse to only three methods: residues(), length(), and metaPos(). This means that CGL users needn't concern themselves with the details of the data-model or database schema associated with the annotations. Nor need they concern themselves with the guts of CGL. This means that very little time is required to master CGL, freeing users to focus on biological problems, rather than gory programming details.

Every sequence alignment implicitly aligns two genes: using Phat Hits & HSPs

CGL provides the means to compare genome annotations to one another using BLAST sequence alignments. It does so by extending the BioPerl GenericHit and GenericHSP classes. These extensions, called PhatHit and PhatHSP, provide additional methods that greatly facilitate comparative genomics as they grant users improved access to the coordinate relationships implied by BLAST HSPs. Whereas the existing BioPerl classes provide access only to the begin and end coordinates of an alignment, the Phat classes provide access to the explicit and implied positions of any aligned residue in a BLAST HSP. Their methods operate on gapped alignments. Moreover, the Phat classes also transparently manage the complexities that arise when attempting to relate the position of an aligned amino acid in a BLASTX, TBLASTN, or TBLASTX alignment to its actual position on the nucleotide sequence provided to BLAST. Requests, for example, for the position on an aligned amino acid in the query portion of a TBLASTN alignment will return the position of that amino acid on the protein query, whereas a request for a position on the sbjct portion for the alignment are returned relative to the subject's nucleotide sequence. Perhaps, more importantly, once a user has identified a position of interest on one strand of an alignment, the equivalent coordinate on the other sequence can be recovered in a single method call. This feature greatly facilitates comparative genomics analyses, as that coordinate can then be passed to CGL's metaPos() method allowing users to explore other, implied relationships between the parts of two genome annotations.

Parsing a BLAST report using the Phat Classes

To employ the CGL extensions of the Bioperl Hit and HSP classes when parsing a BLAST report first create a Bioperl Bio::SearchIO object:

```
$sio = new Bio::SearchIO(-format => 'blast', -file => 'my_blast_report').
```

The next step is to select the appropriate subclass of PhatHit and PhatHSP objects to employ. Use the blastn

subclasses for BLASTN reports, the tblastx subclasses when parsing TBLASTX reports, etc. If parsing a BLASTP report, for example, use the blastp subclasses:

```
$hit_type = 'Bio::Search::Hit::GenericHit::blastp::PhatHit';  
  
$hsp_type = 'Bio::Search::Hit::GenericHit::blastp::PhatHSP'.
```

The next step is to create Bioperl Object factories that will populate these classes when parsing the blast report:

```
$hit_factory = new Bio::Search::Hit::HitFactory(-type => $hit_type);  
  
$hsp_factory = new Bio::Search::HSP::HSPFactory(-type => $hsp_type);
```

These object factories are then registered with the Bioperl SearchIO event handler as follows.

```
$sio->_eventHandler->register_factory('hit', $hit_factory).  
  
$sio->_eventHandler->register_factory('hsp', $hsp_factory).
```

Hit and HSP objects are now employed just as they would be normally using Bioperl e.g.

```
$result = $sio->next_result();  
  
$hit = $result->next_hit();  
  
$hsp = $hit->next_hsp();
```

But, because they are Phat, they provide an extended set of methods designed to facilitate comparative genomics. The most important of these methods are documented below.

Using Phat HSPs together with annotations for comparative genomics

PhatHSPs are extensions of the Bioperl GenericHSP class. For working examples of how to use Phat Hits and HSPs see *cgl_phat_tutorial*, a starter script located in the CGL *script* sub-directory.

For purposes of the following description of their use, consider the BLASTP HSP shown in figure 2. In the examples below, the CGL object corresponding to the query protein aligned in figure 2, is denoted as \$q_p; that protein's transcript, is \$q_t; and one of its exons is \$q_e. Likewise, \$s_p denotes the corresponding sbjct protein, and \$s_t that protein's corresponding transcript.

```
Score = 94 (38.1 bits), Expect = 2.2e-10, P =  
2.2e-10, Identities = 15/18 (83%),  
Positives = 16/18 (88%)
```

```
Query:   1 MDEWRQWKMD+QVWDAER 18  
          MD+WR WKMD+QVWD ER  
Sbjct:  1 MDDWR-WKMDKQVWDMER 17
```

Figure 2. A WU-BLASTP HSP

Suppose one is interested in determining if the sequence alignment shown in figure 2 implies that the two genes producing those proteins possess an intron at the same position relative to their protein sequences.

The position of a particular splice junction on the query protein can be obtained with the CGL call:

```
$pos_on_q_protein = $q_t->metaPos($q_p, $q_e->metaPos($q_t, 0)).
```

A value of, for example, 10 would mean that the exon, \$q_e, began in phase 0, just prior to the 'E' located at position 11 of query sequence shown figure 2. For more information about how CGL handles phase relationships see the preceding section entitled 'Access to Explicit and Implicit Coordinate Relationships'. Note that coordinate returned by CGL's metaPos() method must be incremented by 1 as unlike CGL, Bioperl does not employ a zero, space-based coordinate system.

Of course the fact that a splice junction maps to a position on the protein, says nothing about whether or not that region of the protein is aligned a particular HSP. To ascertain this fact use the two calls shown below.

```
next unless $hsp->nB('query') <= $pos_on_q_protein + 1.
```

```
next unless $hsp->nE('query') >= $pos_on_q_protein + 1.
```

PhatHSPs provide two methods called nB() & nE() or natural begin and natural end, respectively. These method behave exactly like the Bioperl GenericHSP start() and end() methods, except that they preserve the native order of the start and end coordinates of an HSP given in the BLAST report, i.e.,

`$hsp->nB() > $hsp->nE()` for minus strand hits.

These methods facilitate some analyses. Note that PhatHit and PhatHSP objects observe the Bioperl practice of referring to the 'subjct' sequence of a BLAST HSP as the 'hit'.

Alternatively, whether or not an HSP contains a position of interest in its alignment can also be determined with the call:

```
$hsp->isContained('query', $pos_on_q_protein + 1).
```

The isContained() method will return 0 if the position on the specified 'query' or 'hit' sequence lies outside of the HSP or if the residue located at the position specified by the 2nd argument is opposite a gap in its partner, otherwise it will return 1.

Phat HSPs provide easy access to the character present at any position of a BLAST sequence alignment through the whatIsThere() method; the first argument to this method specifies the sequence of interest: 'query' or 'hit', the second the position of interest on that sequence.

```
$hsp->whatIsThere('query', $pos_on_q_protein + 1).
```

For the HSP shown in figure 2, the call shown above will return an 'E', as this is the amino acid present at that position of the sequence alignment in figure 2.

The equivalent position on the subjct sequence can be recovered with the following call.

```
$hsp->equivalent_pos_in_alignment_partner('query', $pos_on_q_protein + 1).
```

The first argument to the above method denotes which sequence, 'query' or 'hit' the coordinate given as the second argument lies on. The method returns the equivalent position on the alignment partner. If \$hsp were a TBLASTN or TBLASTX HSP, the return value would in nucleotide coordinates. In this example, the method would return a value of 10 (see figure 2), as this position on the hit sequence that corresponds to a position of 11 on the query sequence. Note that the equivalent position on the annotated hit protein is 9. (recall that BioPerl counts from base 1, whereas CGL counts from space 0) A fact that can be verified with the call:

```
die unless $hsp->whatIsThere('hit', 10) eq substr($s_p->residues, 9, 1),
```

where \$s_p is the corresponding CGL protein object for the subjct sequence. Both calls would return a

'K' for the example shown in figure 2. Had the residue at that position on the query sequence been opposite a gap in the subj sequence, the method would have returned undefined.

Determining if the gene producing the subject protein contains an intron at this position is accomplished with a short CGL based subroutine, e.g.,

```
print "YES\n." if intron_at_this_pos_on_protein($s_t, $s_p, 9);
sub intron_at_this_pos_on_protein {
    ($t, $p, $pos) = @_;
    $i= 0;
    while($e = $t->exon($i)){
        $i++;
        $i_pos = $t->metaPos($p, $e->metaPos($t, 0));
        next unless defined($i_pos);
        return 1 if int($i_pos) == $pos;
    }
    return 0;
}
```

Obviously many variations of the examples shown in this brief exposition are possible. PhatHSPs and the protein, transcript and exon objects provided by CGL are designed to complement one another, especially as regards the metaPos() method. The ability to easily relate the parts of one annotation to those of another using a sequence alignment makes simple many analyses that were previously next to impossible.

Conclusions

The analyses presented here demonstrate that CGL provides an easy means to explore, compare, and characterize genome annotations. As the code will run anywhere the CGL library is installed and Chaos documents are available, changing workplaces does not longer require downtime in order to master the quirks of the local database schema and programming environment; all that's required to use CGL is a basic knowledge of PERL, and familiarity with concepts such as exon, transcript and protein.

For more information about using CGL for analyses see:

Large-Scale Trends in the Evolution of Gene Structures within 11 Animal Genomes

Mark Yandell, Chris J. Mungall, Chris Smith, Simon Prochnik, Joshua Kaminker, George Hartzell, Suzanna Lewis & Gerald M. Rubin. PloS Computational Biology, in press.

A computational and experimental approach to validating annotations and gene predictions in the *Drosophila melanogaster* genome. Mark Yandell, Adina M. Bailey, Sima Misra, ShengQiang Shu, Colin Wiel, Martha Evans-Holm, Susan E. Celniker and Gerald M. Rubin
PNAS, February 1, 2005; vol. 102; no. 5;pp. 1566-1571

Appendix 1: Details of the Chaos data-model

Please see the following web page: www.fruitfly.org/chaos-xml

Appendix 2: CGL is based around the Sequence Ontology

Comparing the parts of genes to one-another dictates that the terms used to describe an annotation and its parts be precisely defined; note that 'parts' means things like 'exon', intron 'stop codon', etc. Normally, one relies on common usage when applying descriptive terms such as 'exon', or 'UTR', to a gene, but large-scale comparative genomics requires that we be more precise, lest ambiguity creep into comparisons. Does a UTR contain a stop codon? Do all transcripts contain exons? Rational arguments can be made for answering either of these questions with a 'yes' or 'no'. What is important is that there exist a fixed and fully defined terminology for describing the parts of an annotation and how they relate to one another. It is for these reasons that CGL employs the Sequence Ontology [2].

CGL employs the Sequence Ontology to define its class schema, inheritance hierarchy and interfaces. The Current release of CGL contains modules for contigs, genes, transcripts, exons, introns, and protein objects. These objects are created on demand when CGL parses a Chaos XML document. For example, transcript objects are created when CGL encounters the term 'transcript' in a document. Likewise, because the Sequence Ontology describes an exon as a legitimate part of a transcript, CGL will look for, automatically create and attach the appropriate set of exon objects to each transcript object.

CGL also uses the Sequence Ontology as a natural paradigm for sub-classing objects. For example, if a portion of an annotation is described as an 'mRNA' in a document, CGL will create a transcript object corresponding to that feature; this is because an 'mRNA' *'isa'* 'Transcript' according to the Sequence Ontology. Thus, any term that is a child of the Sequence Ontology terms 'Transcript', 'Exon', 'Intron', and 'Protein' can be used to describe the parts of a gene in a Chaos XML document.

The methods appropriate to a class are also (in part) dictated by the Sequence Ontology. If an mRNA is found to have a defining, essential part, then at some future point in time, it may prove advisable to create a CGL::Annotation::Feature::Transcript::mRNA class containing a method particular to an mRNA.. Thus, the Sequence Ontology also provides a natural direction in which to extend CGL as it matures.

Bibliography

1. Lewis SE, Searle SMJ, Harris N. *et al.*, 2002. Apollo: a sequence annotation editor. *Genome Biology* 3(12):research 0082
2. Eilbeck K, Lewis SE, Mungall CJ, Yandell M, Stein L, et al. (2005) The Sequence Ontology: a tool for the unification of genome annotations. *Genome Biol* 6: R44.
3. Generic Model Organism Database [www.gmod.org]
4. Stajich JE, Block D, Boulez K, Brenner SE, Chervitz SA, Dagdigian C, Fuellen G, Gilbert JG, Korf I, Lapp H *et al*: The Bioperl toolkit: Perl modules for the life sciences. *Genome Res* 2002, **12**(10):1611-1618.
5. Benson DA, Karsch-Mizrachi I, Lipman DJ, Ostell J, DL. 2003. GenBank *Nucleic Acids Res.* 31: 23-27.
6. Ensembl homepage [www.ebi.ac.uk/ensembl/] Dwight SS, Balakrishnan R, Christie KR, Costanzo MC, Dolinski K, Engel SR, Feierbach B, Fisk DG, Hirschman J, Hong EL et al.
7. *Saccharomyces* genome database: underlying principles and organisation. *Brief Bioinform.* 2004,**5**:9-22.
8. Misra S, Crosby MA, Mungall CJ, Matthews BB, Campbell KS, Hradecky P, Huang Y, Kamiker JS, Millburn GH, Prochnik SE et al.: Annotation of the *Drosophila melanogaster* euchromatic genome: a systematic review. *Genome Biology* 2002, **3**:RESEARCH0083.1-0083.22
9. Stein L, Sternberg P, Durbin R, Thierry-Mieg J, Spieth J: WormBase: network access to the genome and biology of *Caenorhabditis elegans*. *Nucleic Acids Res.* 2001, **29**:82-86.