- BioPerl Tutorial
- I. Introduction
  - I.1 Overview
  - I.2 Software requirements
  - I.2.1 Minimal bioperl installation
  - I.2.2 Complete installation
  - I.3 Installation
  - I.4 Additional comments for non-unix users
- II. Brief introduction to bioperl's objects
  - II.2 Sequence objects: (Seq, PrimarySeq, LocatableSeq, LiveSeq, LargeSeq, SeqI)
  - II.3 Alignment objects (SimpleAlign, UnivAln)
  - II.4 Interface objects and implementation objects
- III. Using bioperl
  - III.1 Accessing sequence data from local and remote databases
  - III.1.1 Accessing remote databases (Bio::DB::GenBank, etc)
  - III.1.2 Indexing and accessing local databases (Bio::Index::*,
  - III.2 Transforming formats of database/ file records
  - III.2.1 Transforming sequence files (SeqIO)
  - III.2.2 Transforming alignment files (AlignIO)
  - III.3 Manipulating sequences
  - III.3.1 Manipulating sequence data with Seq methods
  - III.3.2 Obtaining basic sequence statistics- MW, residue &codon
  - III.3.3 Identifying restriction enzyme sites (RestrictionEnzyme)
  - III.3.4 Identifying amino acid cleavage sites (Sigcleave)
  - III.3.5 Miscellaneous sequence utilities: OddCodes, SeqPattern
  - III.4 Searching for "similar" sequences
  - III.4.1 Running BLAST locally (StandAloneBlast)
  - III.4.2 Running BLAST remotely (using Blast.pm)
  - III.4.3 Parsing BLAST reports with Blast.pm
  - III.4.4 Parsing BLAST reports with BPlite, BPpsilite and BPbl2seq
  - III.4.5 Parsing HMM reports (HMMER::Results)
  - III.5 Creating and manipulating sequence alignments
  - III.5.1 Aligning 2 sequences with Smith-Waterman (pSW)
  - III.5.2 Aligning 2 sequences with Blast using bl2seq and AlignIO
  - III.5.3 Aligning multiple sequences (Clustalw.pm, TCoffee.pm)
  - III.5.4 Manipulating / displaying alignments (SimpleAlign, UnivAln)
  - III.6 Searching for genes and other structures on genomic DNA
  - III.7 Developing machine readable sequence annotations
  - III.7.1 Representing sequence annotations (Annotation,
  - III.7.2 Representing and large and/or changing sequences
  - III.7.3 Representing related sequences - mutations,
  - III.7.4 Sequence XML representations - generation and parsing
- IV. Related projects - biocorba, biopython, biojava, Ensembl,
  - IV.1 Biocorba
  - IV.2 Biopython and biojava
  - IV.3 Ensembl
  - IV.4 The Annotation Workbench and bioperl-gui
  - V.1 Appendix: Public Methods of Bioperl Objects

- V.2 Appendix: Tutorial demo scripts

BioPerl Tutorial

Cared for by Peter Schattner <schattner@alum.mit.edu>

Copyright Peter Schattner

 This tutorial includes "snippets" of code and text from various
 Bioperl documents including module documentation, example scripts
 and "t" test scripts. You may distribute this tutorial under the
 same terms as perl itself.

 This document is written in Perl POD (plain old documentation) format.
 You can run this file through your favorite pod translator (pod2html, pod2man,
 pod2text, etc.) if you would like a more convenient formatting.

Table of Contents

I. Introduction
I.1 Overview
I.2 Software requirements
  I.2.1 For minimal bioperl installation
  I.2.2 For complete installation
I.3 Installation procedures
I.4 Additional comments for non-unix users

II. Brief overview to bioperl’s objects
II.1 Sequence objects: (Seq, PrimarySeq, LocatableSeq, LiveSeq, LargeSeq, SeqI)
II.2 Alignment objects (SimpleAlign, UnivAln)
II.3 Interface objects and implementation objects

III. Using bioperl
III.1 Accessing sequence data from local and remote databases
   III.1.1 Accessing remote databases (Bio::DB::GenBank, etc)
   III.1.2 Indexing and accessing local databases (Bio::Index::*, bpindex.pl, bp
III.2 Transforming formats of database/ file records
   III.2.1 Transforming sequence files (SeqIO)
   III.2.2 Transforming alignment files (AlignIO)
III.3 Manipulating sequences
  III.3.1 Manipulating sequence data with Seq methods (Seq)
  III.3.2 Obtaining basic sequence statistics- MW, residue &codon frequencies (Seq
  III.3.3 Identifying restriction enzyme sites (RestrictionEnzyme)
  III.3.4 Identifying amino acid cleavage sites (Sigcleave)
  III.3.5 Miscellaneous sequence utilities: OddCodes, SeqPattern
III.4 Searching for "similar" sequences
   III.4.1 Running BLAST locally (StandAloneBlast)
   III.4.2 Running BLAST remotely (using Blast.pm)
   III.4.3 Parsing BLAST reports with Blast.pm
   III.4.4 Parsing BLAST reports with BPlite, BPpsilite and BPbl2seq
   III.4.5 Parsing HMM reports (HMMER::Results)
III.5 Creating and manipulating sequence alignments
   III.5.1 Aligning 2 sequences with Smith-Waterman (pSW)
   III.5.2 Aligning 2 sequences with Blast using bl2seq and AlignIO
   III.5.3 Aligning multiple sequences (Clustalw.pm, TCoffee.pm)

   III.5.4 Manipulating / displaying alignments (SimpleAlign, UnivAln)
III.6 Searching for genes and other structures on genomic DNA (Genscan, Sim4, ESTS
III.7 Developing machine readable sequence annotations
   III.7.1 Representing sequence annotations (Annotation, SeqFeature)
   III.7.2 Representing and large and/or changing sequences (LiveSeq,LargeSeq)
   III.7.3 Representing related sequences - mutations, polymorphisms etc (Allele,
   III.7.4 Sequence XML representations - generation and parsing (SeqIO::game)

IV. Related projects - biocorba, biopython, biojava, Ensembl, AnnotationWorkbench
   IV.1 Biocorba
   IV.2 Biopython and biojava
   IV.3 Ensembl
   IV.4 The Annotation Workbench and bioperl-gui

V. Appendices
   V.1 Public Methods of Principal Bioperl Objects
   V.2 Tutorial Demo Scripts

I. Introduction

I.1 Overview

Bioperl is a collection of perl modules that facilitate the development of perl scripts for bio-informatics applications. As such, it does not include ready to use programs in the sense that may commercial packages and free web-based interfaces (eg Entrez, SRS) do. On the other hand, bioperl does provide reusable perl modules that facilitate writing perl scripts for sequence manipulation, accessing of databases using a range of data formats and execution and parsing of the results of various molecular biology programs including Blast, clustalw, TCoffee, genscan, ESTscan and HMMER. Consequently, bioperl enables developing scripts that can analyze large quantities of sequence data in ways that are typically difficult or impossible with web based systems.

In order to take advantage of bioperl, the user needs a basic understanding of the perl programming language including an understanding of how to use perl references, modules, objects and methods. If these concepts are unfamiliar the user is referred to any of the various introductory / intermediate books on perl. (I’ve liked S. Holzmer’s Perl Core Language, Coriolis Technology Press, for example). This tutorial is not intended to teach the fundamentals of perl to those with little or no experience in the perl language. On the other hand, advanced knowledge of perl - such as how to write a perl object - is not required for successfully using bioperl.

Bioperl is open source software that is still under active development. The advantages of open source software are well known. They include the ability to freely examine and modify source code and exemption from software licensing fees. However, since open source software is typically developed by a large number of volunteer programmers, the resulting code is often not as clearly organized and its user interface not as standardized as in a mature commercial product. In addition, in any project under active development, documentation may not keep up with the development of new features. Consequently the learning curve for actively developed, open source source software is sometimes

steep.

This tutorial is intended to ease the learning curve for new users of bioperl. To that end the tutorial includes:

- Descriptions of what bio-informatics tasks can be handled with bioperl
- Directions on where to find the methods to accomplish these tasks within the bioperl package
- Recommendations on where to go for additional information.
- A separate tutorial script (tutorial.pl - located in the top bioperl directory) with examples of many of methods described in the tutorial.

Running the tutorial.pl script while going through this tutorial - or better yet, stepping through it with an interactive debugger - is a good way of learning bioperl. The tutorial script is also a good place from which to cut-and-paste code for your `scripts(rather` than using the code snippets in this tutorial). The tutorial script should work on your machine - and if it doesn't it would probably be a good idea to find out why, before getting too involved with bioperl!

This tutorial does not intend to be a comprehensive description of all the objects and methods available in bioperl. For that the reader is directed to the documentation included with each of the modules as well as the additional documentation referred to below.

I.2 Software requirements

I.2.1 Minimal bioperl installation

For a ‘‘minimal’’ installation of bioperl, you will need to have perl itself installed as well as the bioperl ‘‘core modules’’. Bioperl has been tested primarily using perl 5.005 and more recently perl 5.6. The minimal bioperl installation should still work under perl 5.004. However, as increasing numbers of bioperl objects are using modules from CPAN (see below), problems have been observed for bioperl running under perl 5.004. So if you are having trouble running bioperl under perl 5.004, you should probably upgrade your version of perl.

In addition to a current version of perl, the new user of bioperl is encouraged to have access to, and familiarity with, an interactive perl debugger. Bioperl is a large collection of complex interacting software objects. Stepping through a script with an interactive debugger is a very helpful way of seeing what is happening in such a complex software system - especially when the software is not behaving in the way that you expect. The free graphical debugger ptkdb (available as Devel::ptkdb from CPAN) is highly recommended. Active State offers a commercial graphical debugger for windows systems. The standard perl distribution also contains a powerful interactive debugger - though with a more cumbersome (command line) interface.

I.2.2 Complete installation

Taking full advantage of bioperl requires software beyond that for the minimal installation. This additional software includes perl modules from CPAN, bioperl perl extensions, a bioperl xs-extension, and several standard compiled bioinformatics programs.

```
Perl - extensions
```

The following perl modules are available from bioperl (http://bioperl.org/Core/external.shtml)or from CPAN (http://www.perl.com/CPAN/) are used by bioperl. The listing also indicates what bioperl features will not be available if the corresponding CPAN module is not downloaded. If these modules are not available (eg non-unix operating systems), the remainder of bioperl should still function correctly.

For accessing remote databases you will need:

- File-Temp-0.09
- IO-String-1.01

For accessing Ace databases you will need:

- AcePerl-1.68.

For remote blast searches you will need:

- `libwww-perl-5.48`
- `Digest-MD5-2.12.`
- `HTML-Parser-3.13`

```
=item *
```

- `libnet-1.0703`
- `MIME-Base64-2.11`
- `URI-1.09`
- `IO-stringy-1.216`

For xml parsing you will need:

- libxml-perl-0.07
- XML-Node-0.10
- XML-Parser.2.30
- XML-Writer-0.4
- expat-1.95.1 from http://sourceforge.net/projects/expat/

For more current and additional information on external modules required by bioperl, check http://bioperl.org/Core/external.shtml

```
Bioperl c \extensions & external bio-informatics programs
```

Bioperl also uses several c-programs for sequence alignment and local blast searching. To use these features of bioperl you will need an ANSI C or Gnu C compiler as well as the actual program available from sources such as:

for smith-waterman alignments- bioperl-ext-0.6 from http://bioperl.org/Core/external.shtml

for clustalw alignments- http://corba.ebi.ac.uk/Biocatalog/Alignment_Search_software.html

for tcoffee alignments-
http://igs-server.cnrs-mrs.fr/~cnotred/Projects_home_page/t_coffee_home_page.html

for local blast searching- ftp://ncbi.nlm.nih.gov/blast

I.3 Installation

The actual installation of the various system components is accomplished in the standard manner:

- Locate the package on the network
- Download
- Decompress (with gunzip or a simliar utility)
- Remove the file archive (eg with tar -xvf)
- Create a ‘‘makefile’’ (with ‘‘perl Makefile.PL’’ for perl modules or a supplied ‘‘install’’ or ‘‘configure’’ program for non-perl program
- Run ‘‘make’’, ‘‘make test’’ and ‘‘make install’’ This procedure must be repeated for every CPAN module, bioperl-extension and external module to be installed. A helper module CPAN.pm is available from CPAN which automates the process for installing the perl modules.

For the external programs (clustal, Tcoffee, ncbi-blast), there is an extra step:

- Set the relevant environmental variable (CLUSTALDIR, TCOFFEEDIR or BLASTDIR) to the directory holding the executable in your startup file - eg in .bashrc. (For running local blasts, it is also necessary that the name of local-blast database directory is known to bioperl. This will typically happen automatically, but in case of difficulty, refer to the documentation for StandAloneBlast.pm)

The only likely complication (at least on unix systems) that may occur is if you are unable to obtain system level writing privileges. For instructions on modifying the installation in this case and for more details on the overall installation procedure, see the README file in the bioperl distribution as well as the README files in the external programs you want to use (eg bioperl-ext, clustalw, TCoffee, NCBI-blast).

I.4 Additional comments for non-unix users

Bioperl has mainly been developed and tested under various unix environments (including Linux) and this tutorial is intended primarily for unix users. The minimal installation of bioperl *should* work under other OS's (NT, windows, perl). However, bioperl has not been widely tested under these OS's and problems have been noted in the bioperl mailing lists. In addition, many bioperl features require the use of CPAN modules, compiled extensions or external programs. These features will probably will not work under some or all of these other operating systems. If a script attempts to access these features from a non-unix OS, bioperl is designed to simply report that the desired capability is not available. However, since the testing of bioperl in these environments has been limited, the script may well crash in a less ''graceful'' manner.

Todd Richmond has written of his experiences with BioPerl on MacOs at
http://bioperl.org/Core/mac-bioperl.html

II. Brief introduction to bioperl's objects

The purpose of this tutorial is to get you using bioperl to solve real-life bioinformatics problems as quickly as possible. The aim is not to explain the structure of bioperl objects or perl object-oriented programming in general. Indeed, the relationships among the bioperl objects is not simple; however, understanding them in detail is fortunately not necessary for successfully using the package.

Nevertheless, a little familiarity with the bioperl object ''bestiary'' can be very helpful even to the casual user of bioperl. For example there are (at least) six different ''sequence objects'' - Seq, PrimarySeq, LocatableSeq, LiveSeq, LargeSeq, SeqI. Understanding the relationships among these objects - and why there are so many of them - will help you select the appropriate one to use in your script.

II.2 Sequence objects: (Seq, PrimarySeq, LocatableSeq, LiveSeq, LargeSeq, SeqI)

Seq is the central sequence object in bioperl. When in doubt this is probably the object that you want to use to describe a dna, rna or protein sequence in bioperl. Most common sequence manipulations can be performed with Seq. These capabilities are described in sections III.3.1 and III.7.1.

Seq objects can be created explicitly (see section III.2.1 for an example). However usually Seq objects will be created for you automatically when you read in a file containing sequence data using the SeqIO object. This procedure is described in section III.2.1. In addition to storing its identification labels and the sequence itself, a Seq object can store multiple annotations and associated ''sequence features''. This capability can be very useful - especially in development of automated genome annotation systems, see section III.7.1.

On the other hand, if you need a script capable of simultaneously handling many (hundreds or thousands) sequences at a time, then the overhead of adding annotations to each sequence can be significant. For such applications, you will want to use the PrimarySeq object. PrimarySeq is basically a ‘‘stripped down’’ version of Seq. It contains just the sequence data itself and a few identifying labels (id, accession number, molecule type = dna, rna, or protein). For applications with hundreds or thousands or sequences, using PrimarySeq objects can significantly speed up program execution and decrease the amount of RAM the program requires.

The LocatableSeq object is just a Seq object which has ‘‘start’’ and ‘‘end’’ positions associated with it. It is used by the alignment object SimpleAlign and other modules that use SimpleAlign objects (eg AlignIO, pSW). In general you don’t have to worry about creating LocatableSeq objects because they will be made for you automatically when you create an alignment (using pSW, Clustalw, Tcoffee or bl2seq) or when input an alignment data file using AlignIO. However if you need to input a sequence alignment by hand (ieg to build a SimpleAlign object), you will need to input the sequences as LocatableSeqs.

A LargeSeq object is a special type of Seq object used for handling very long (eg *gt* 100 MB) sequences. If you need to manipulate such long sequences see section III.7.2 which describes LargeSeq objects.

A LiveSeq object is another specialized object for storing sequence data. LiveSeq addresses the problem of features whose location on a sequence changes over time. This can happen, for example, when sequence feature objects are used to store gene locations on newly sequenced genomes - locations which can change as higher quality sequencing data becomes available. Although a LiveSeq object is not implemented in the same way as a Seq object, LargeSeq does implement the SeqI interface (see below). Consequently, most methods available for Seq objects will work fine with LiveSeq objects. Section III.7.2 contains further discussion of LiveSeq objects.

SeqI objects are Seq ‘‘interface objects’’ (see section II.4) They are used to ensure bioperl’s compatibility with other software packages. SeqI and other interface objects are not likely to be relevant to the casual bioperl user.

*** Having described these other types of sequence objects, the ‘‘bottom line’’ still is that if you store your sequence data in Seq objects (which is where they’ll be if you read them in with SeqIO), you will usually do just fine. ***

II.3 Alignment objects (SimpleAlign, UnivAln)

There are two ‘‘alignment objects’’ in bioperl: SimpleAlign and UnivAln. Both store an array of sequences as an alignment. However their internal data structures are quite different and converting between them - though certainly possible - is rather awkward. In contrast to the sequence objects - where there are good reasons for having 6 different classes of objects, the presence of two alignment objects is just an unfortunate relic of the two systems having been designed independently at different times.

Since each object has some capabilities that the other lacks it has not yet been feasible to unify bioperl’s sequence alignment methods into a single object (see section III.5.4 for a description of SimpleAlign’s

and UnivAln's features) . However, recent development in bioperl involving alignments has been focused on using SimpleAlign and the new user should generally use SimpleAlign where possible.

II.4 Interface objects and implementation objects

Since release 0.6, bioperl has been moving to separate interface and implementation objects. An interface is solely the definition of what methods one can call on an object, without any knowledge of how it is implemented. An implementation is an actual, working implementation of an object. In languages like Java, interface definition is part of the language. In Perl, you have to roll your own.

In bioperl, the interface objects usually have names like Bio::MyObjectI, with the trailing I indicating it is an interface object. The interface objects mainly provide documentation on what the interface is, and how to use it, without any implementations (though there are some exceptions). Although interface objects are not of much direct utility to the casual bioperl user, being aware of their existence is useful since they are the basis to understanding how bioperl programs can communicate with other bioinformatics projects such as Ensembl and the Annotation Workbench (see section IV)

III. Using bioperl

Bioperl provides software modules for many of the typical tasks of bioinformatics programming. These include:

- **Accessing sequence data from local and remote databases**
- **Transforming formats of database/ file records**
- **Manipulating individual sequences**
- **Searching for "similar" sequences**
- **Creating and manipulating sequence alignments**
- **Searching for genes and other structures on genomic DNA**
- **Developing machine readable sequence annotations**

The following sections describe how bioperl can help perform all of these tasks.

III.1 Accessing sequence data from local and remote databases

Much of bioperl is focused on sequence manipulation. However, before bioperl can manipulate sequences, it needs to have access to sequence data. Now one can directly enter data sequence data into a bioperl Seq object, eg:

```
$seq = Bio::Seq->new('-seq'=>'actgtggcgtcaact',
                     '-desc'=>'Sample Bio::Seq object',
```

```
    '-display_id' => 'something',
    '-accession_number' => 'accnum',
    '-moltype' => 'dna' );
```

However, in most cases, it is preferable to access sequence data from some online data file or database (Note that in common with conventional bioinformatics usage we will call a ‘‘database’’ what might be more appropriately referred to as an ‘‘indexed flat file’’.) Bioperl supports accessing remote databases as well as developing indices for setting up local databases.

III.1.1 Accessing remote databases (Bio::DB::GenBank, etc)

Accessing sequence data from the principal molecular biology databases is straightforward in bioperl. Data can be accessed by means of the sequence's accession number or id. Batch mode access is also supported to facilitate the efficient retrieval of multiple sequences. For retrieving data from genbank, for example, the code could be as follows:

```
$gb = new Bio::DB::GenBank();
$seq1 = $gb->get_Seq_by_id('MUSIGHBA1');
$seq2 = $gb->get_Seq_by_acc('AF303112'))
$seqio = $gb->get_Stream_by_batch([ qw(J00522 AF303112 2981014)]));
```

Bioperl currently supports sequence data retrieval from the genbank, genpept, swissprot and gdb databases. Bioperl also supports retrieval from a remote Ace database. This capability requires the presence of the external AcePerl module. You need to download and install the aceperl module from http://stein.cshl.org/AcePerl/.

III.1.2 Indexing and accessing local databases (Bio::Index::*, bpindex.pl, bpfetch.pl)

Alternately, bioperl permits indexing local sequence data files by means of the Bio::Index objects. The following sequence data formats are supported: genbank, swissprot, pfam, embl and fasta. Once the set of sequences have been indexed using Bio::Index, individual sequences can be accessed using syntax very similar to that described above for accessing remote databases. For example, if one wants to set up an indexed (flat-file) database of fasta files, and later wants then to retrieve one file, one could write a scripts like:

```
# script 1: create the index
use Bio::Index::Fasta; # using fasta file format
$Index_File_Name = shift;
$inx = Bio::Index::Fasta->new(
    -filename => $Index_File_Name,
    -write_flag => 1);
$inx->make_index(@ARGV);

# script 2: retrieve some files
use Bio::Index::Fasta;
$Index_File_Name = shift;
```

```
$inx = Bio::Index::Fasta->new($Index_File_Name);
foreach  $id (@ARGV) {
    $seq = $inx->fetch($id);  # Returns Bio::Seq object
    # do something with the sequence
}
```

To facilitate the creation and use of more complex or flexible indexing systems, the bioperl distribution includes two sample scripts bpindex.pl and bpfetch.pl. These scripts can be used as templates to develop customized local data-file indexing systems.

III.2 Transforming formats of database/ file records

III.2.1 Transforming sequence files (SeqIO)

A common - and tedious - bioinformatics task is that of converting sequence data among the many widely used data formats. Bioperl's SeqIO object, however, makes this chore a breeze. SeqIO can read a stream of sequences - located in a single or in multiple files - in any of six formats: Fasta, EMBL. GenBank, Swissprot, PIR and GCG. Once the sequence data has been read in with SeqIO, it is available to bioperl in the form of Seq objects. Moreover, the Seq objects can then be written to another file (again using SeqIO) in any of the supported data formats making data converters simple to implement, for example:

```
use Bio::SeqIO;
$in  = Bio::SeqIO->new('-file' => "inputfilename",
                       '-format' => 'Fasta');
$out = Bio::SeqIO->new('-file' => ">outputfilename",
                       '-format' => 'EMBL');
while ( my $seq = $in->next_seq() ) {$out->write_seq($seq); }
```

In addition, perl ‘‘tied filehandle’’ syntax is available to SeqIO, allowing you to use the standard <> and print operations to read and write sequence objects, eg:

```
$in  = Bio::SeqIO->newFh('-file' => "inputfilename" ,
                         '-format' => 'Fasta');
$out = Bio::SeqIO->newFh('-format' => 'EMBL');
print $out $_ while <$in>;
```

III.2.2 Transforming alignment files (AlignIO)

Data files storing multiple sequence alignments also appear in varied formats. AlignIO is the bioperl object for data conversion of alignment files. AlignIO is patterned on the SeqIO object and shares most of SeqIO's features. AlignIO currently supports input in the following formats: fasta, mase, stockholm, prodom, selex, bl2seq, msf/gcg and output in these formats: : fasta, mase, selex, clustalw, msf/gcg. One significant difference between AlignIO and SeqIO is that AlignIO handles IO for only a single

alignment at a time (SeqIO.pm handles IO for multiple sequences in a single stream.) Syntax for AlignIO is almost identical to that of SeqIO: use Bio::AlignIO;

```
$in  = Bio::AlignIO->new('-file' => "inputfilename" ,
                         '-format' => 'fasta');
$out = Bio::AlignIO->new('-file' => ">outputfilename",
                         '-format' => 'pfam');
while ( my $aln = $in->next_aln() ) { $out->write_aln($aln);  }
```

The only difference is that here, the returned object reference, $aln, is to a SimpleAlign object rather than a Seq object.

AlignIO also supports the tied filehandle syntax described above for SeqIO. (Note that currently AlignIO is usable only with SimpleAlign alignment objects. IO for UnivAln objects can only be done for files in fasta data format.)

III.3 Manipulating sequences

III.3.1 Manipulating sequence data with Seq methods

OK, so we know how to retrieve sequences and access them as Seq objects. Let's see how we can use the Seq objects to manipulate our sequence data and retrieve information. Seq provides multiple methods for performing many common (and some not-so-common) tasks of sequence manipulation and data retrieval. Here are some of the most useful:

The following methods return strings

```
$seqobj->display_id(); # the human read-able id of the sequence
$seqobj->seq();        # string of sequence
$seqobj->subseq(5,10); # part of the sequence as a string
$seqobj->accession_number(); # when there, the accession number
$seqobj->moltype();    # one of 'dna','rna','protein'
$seqobj->primary_id(); # a unique id for this sequence irregardless
                       # of its display_id or accession number
```

The following methods return an array of Bio::SeqFeature objects

```
 $seqobj->top_SeqFeatures # The 'top level' sequence features
 $seqobj->all_SeqFeatures # All sequence features, including sub
                          # seq features
```

Sequence features will be discussed further in section III.7 on machine-readable sequence annotation.

The following methods returns new sequence objects, but do not transfer features across

```
$seqobj->trunc(5,10)  # truncation from 5 to 10 as new object
$seqobj->revcom       # reverse complements sequence
$seqobj->translate    # translation of the sequence
```

Note that some methods return strings, some return arrays and some return references to objects. Here (as elsewhere in perl and bioperl) it is the user's responsibility to check the relevant documentation so they know the format of the data being returned.

Many of these methods are self-explanatory. However, bioperl's flexible translation methods warrant further comment. Translation in bioinformatics can mean two slightly different things:

1. .

   Translating a nucleotide sequence from start to end.
2. .

   Taking into account the constraints of real coding regions in mRNAs.

For historical reasons the bioperl implementation of translation does the first of these tasks easily. Any sequence object which is not of moltype 'protein' can be translated by simply calling the method which returns a protein sequence object:

```
$translation1 = $my_seq_object->translate;
```

However, the translate method can also be passed several optional parameters to modify its behavior. For example, the first two arguments to ''translate'' can be used to modify the characters used to represent stop (default '*') and unknown amino acid ('X'). (These are normally best left untouched.) The third argument determines the frame of the translation. The default frame is ''0''. To get translations in the other two forward frames, we would write:

```
$translation2 = $my_seq_object->translate(undef,undef,1);
$translation3 = $my_seq_object->translate(undef,undef,2);
```

The fourth argument to ''translate'' makes it possible to use alternative genetic codes. There are currently 16 codon tables defined, including tables for 'Verterbate Mitochondrial', 'Bacterial', 'Alternative Yeast Nuclear' and 'Ciliate, Dasycladacean and Hexamita Nuclear' translation. These tables are located in the object Bio::Tools::CodonTable which is used by the translate method. For example, for mitochondrial translation:

```
$human_mitochondrial_translation =
    $my_seq_object->translate(undef,undef,undef, 2);
```

If we want to translate full coding regions (CDS) the way major nucleotide databanks EMBL, GenBank and DDBJ do it, the translate method has to perform more tricks. Specifically, 'translate' needs to confirm that the sequence has appropriate start and terminator codons at the beginning and the end of the sequence and that there are no terminator codons present within the sequence. In addition, if the genetic code being used has an atypical (non-ATG) start codon, the translate method needs to convert the initial amino acid to methionine. These checks and conversions are triggered by setting the fifth argument of the translate method to evaluate to ''true''.

If argument 5 is set to true and the criteria for a proper CDS are not met, the method, by default, issues a warning. By setting the sixth argument to evaluate to ''true'', one can instead instruct the program to die if an improper CDS is found, e.g.

```
$protein_object =
    $cds->translate(undef,undef,undef,undef,1,'die_if_errors');
```

III.3.2 Obtaining basic sequence statistics- MW, residue &codon frequencies(SeqStats, SeqWord)

In addition to the methods directly available in the Seq object, bioperl provides various ‘‘helper’’ objects to determine additional information about a sequence. For example, the SeqStats object provides methods for obtaining the molecular weight of the sequence as well the number of occurrences of each of the component residues (bases for a nucleic acid or amino acids for a protein.) For nucleic acids, SeqStats also returns counts of the number of codons used. For example:

```
use SeqStats
$seq_stats  =  Bio::Tools::SeqStats->new($seqobj);
$weight = $seq_stats->get_mol_wt();
$monomer_ref = $seq_stats->count_monomers();
$codon_ref = $seq_stats->count_codons();  # for nucleic acid sequence
```

Note: sometimes sequences will contain ‘‘ambiguous’’ codes. For this reason, `get_mol_wt()` returns (a reference to) a two element array containing a greatest lower bound and a least upper bound of the molecular weight.

The SeqWords object is similar to SeqStats and provides methods for calculating frequencies of ‘‘words’’ (eg tetramers or hexamers) within the sequence.

III.3.3 Identifying restriction enzyme sites (RestrictionEnzyme)

Another common sequence manipulation task for nucleic acid sequences is locating restriction enzyme cutting sites. Bioperl provides the RestrictionEnzyme object for this purpose. Bioperl’s standard RestrictionEnzyme object comes with data for XXX different restriction enzymes. A list of the available enzymes can be accessed using the `available_list()` method. For example to select all available enzymes that with cutting patterns that are six bases long one would write:

```
$re  = new Bio::Tools::RestrictionEnzyme('-name'=>'EcoRI');
@sixcutters = $re->available_list(6);
```

Once an appropriate enzyme has been selected, the sites for that enzyme on a given nucleic acid sequence can be obtained using the `cut_seq()` method. The syntax for performing this task is:

```
$re1 = new Bio::Tools::RestrictionEnzyme(-name=>'EcoRI');
# $seqobj is the Seq object for the dna to be cut
@fragments =  $re1->cut_seq($seqobj);
```

Adding an enzyme not in the default list is easily accomplished:

```
$re2 = new Bio::Tools::RestrictionEnzyme('-NAME' =>'EcoRV--GAT^ATC',
```

```
                                        '-MAKE' =>'custom');
```

Once the custom enzyme object has been created, `cut_seq()` can be called in the usual manner.

III.3.4 Identifying amino acid cleavage sites (Sigcleave)

For amino acid sequences we may be interested to know whether the amino acid sequence contains a cleavable ‘‘signal sequence’’ for directing the transport of the protein within the cell. SigCleave is a program (originally part of the EGCG molecular biology package) to predict signal sequences, and to identify the cleavage site.

The ‘‘threshold’’ setting controls the score reporting. If no value for threshold is passed in by the user, the code defaults to a reporting value of 3.5. SigCleave will only return score/position pairs which meet the threshold limit.

There are 2 accessor methods for this object. ‘‘signals’’ will return a perl hash containing the sigcleave scores keyed by amino acid position. ‘‘pretty_print’’ returns a formatted string similar to the output of the original sigcleave utility.

Syntax for using the modules is as follows:

```
use Bio::Tools::Sigcleave;
$sigcleave_object = new Bio::Tools::Sigcleave
    ('-file'=>'sigtest.aa',
     '-threshold'=>'3.5'
     '-desc'=>'test sigcleave protein seq',
     '-type'=>'AMINO
    ');
%raw_results      = $sigcleave_object->signals;
$formatted_output = $sigcleave_object->pretty_print;
```

Note that Sigcleave is passed a raw sequence (or file containing a sequence) rather than a sequence object when it is created. Also note that the ‘‘type’’ in the Sigcleave object is ‘‘amino’’ whereas in a Seq object it is ‘‘protein’’.

III.3.5 Miscellaneous sequence utilities: OddCodes, SeqPattern

OddCodes:

For some purposes it’s useful to have a listing of an amino acid sequence showing where the hydrophobic amino acids are located or where the positively charged ones are. Bioperl provides this capability via the module OddCodes.pm.

For example, to quickly see where the charged amino acids are located along the sequence we perform:

```
use Bio::Tools::OddCodes;
```

```
$oddcode_obj = Bio::Tools::OddCodes->new($amino_obj);
$output = $oddcode_obj->charge();
```

The sequence will be transformed into a three-letter sequence (A,C,N) for negative (acidic), positive (basic), and neutral amino acids. For example the ACDEFGH would become NNAANNC.

For a more complete chemical description of the sequence one can call the `chemical()` method which turns sequence into one with an 8-letter chemical alphabet { A (acidic), L (aliphatic), M (amide), R (aromatic), C (basic), H (hydroxyl), I (imino), S (sulfur) }:

```
$output = $oddcode_obj->chemical();
```

In this case the sample sequence ACDEFGH would become LSAARAC.

OddCodes also offers translation into alphabets showing alternate characteristics of the amino acid sequence such as hydrophobicity, ‘‘functionality’’ or grouping using Dayhoff’s definitions. See the documentation for OddCodes.pm for further details.

SeqPattern:

The SeqPattern object is used to manipulate sequences that include perl ‘‘regular expressions’’. A key motivation for SeqPattern is to have a way of generating a reverse complement of a nucleic acid sequence pattern that includes ambiguous bases and/or regular expressions. This capability leads to significant performance gains when pattern matching on both the sense and anti-sense strands of a query sequence are required. Typical syntax for using SeqPattern is shown below. For more information, there are several interesting examples in the script SeqPattern.pl in the examples directory.

```
Use Bio::Tools::SeqPattern;
$pattern     = ’(CCCCT)N{1,200}(agggg)N{1,200}(agggg)’;
$pattern_obj = new Bio::Tools::SeqPattern(’-SEQ’ =>$pattern,
                                          ’-TYPE’ =>’dna’);
$pattern_obj2  = $pattern_obj->revcom();
$pattern_obj->revcom(1); ## returns expanded rev complement pattern.
```

III.4 Searching for "similar" sequences

One of the basic tasks in molecular biology is identifying sequences that are, in some way, similar to a sequence of interest. The Blast programs, originally developed at the NCBI, are widely used for identifying such sequences. Bioperl offers a number of modules to facilitate running Blast as well as to parse the often voluminous reports produced by Blast.

III.4.1 Running BLAST locally (StandAloneBlast)

There are several reasons why one might want to run the Blast programs locally - speed, data security, immunity to network problems, being able to run large batch runs etc. The NCBI provides a downloadable version of blast in a stand-alone format, and running blast locally without any use of perl

or bioperl - is completely straightforward. However, there are situations where having a perl interface for running the blast programs locally is convenient.

The module StandAloneBlast.pm offers the ability to wrap local calls to blast from within perl. All of the currently available options of NCBI Blast (eg PSIBLAST, PHIBLAST, bl2seq) are available from within the bioperl StandAloneBlast interface. Of course, to use StandAloneBlast, one needs to have installed locally ncbi-blast as well as one or more blast-readable databases.

Basic usage of the StandAloneBlast.pm module is simple. Initially, a local blast ‘‘factory object’’ is created.

```
@params = ('program' => 'blastn',
           'database' => 'ecoli.nt');
$factory = Bio::Tools::StandAloneBlast->new(@params);
```

Any parameters not explicitly set will remain as the BLAST defaults. Once the factory has been created and the appropriate parameters set, one can call one of the supported blast executables. The input `sequence(s)` to these executables may be fasta `file(s),` a Bio::Seq object or an array of Bio::Seq objects, eg

```
$input = Bio::Seq->new('-id'=>"test query",
                       '-seq'=>"ACTAAGTGGGGG");
$blast_report = $factory->blastall($input);
```

The returned blast report will be in the form of a bioperl parsed-blast object. The report object may be either a BPlite, BPpsilite, BPbl2seq or Blast object depending on the type of blast search. The ‘‘raw’’ blast report is also available.

The syntax for running PHIBLAST, PSIBLAST and bl2seq searches via StandAloneBlast is also straightforward. See the StandAloneBlast.pm documentation for details. In addition, the script standaloneblast.pl in the examples directory contains descriptions of various possible applications of the StandAloneBlast object.

III.4.2 Running BLAST remotely (using Blast.pm)

Bioperl supports remote execution of blasts at NCBI by means of the Blast.pm object. (Note: the bioperl Blast object is referred to here as Blast.pm to distinguish it from the actual Blast program). Blast.pm is capable of both running Blasts and parsing the report results. Blast.pm supports a wide array of modes, options and parameters. As a result, using Blast.pm directly can be somewhat complicated. Consequently, it is recommended to use, and if necessary modify, the supplied scripts - run_blast_remote.pl and retrieve_blast.pl in the examples/blast/ subdirectory - rather than to use Blast.pm directly. Sample syntax looks like this:

```
run_blast_remote.pl seq/yel009c.fasta -prog blastp -db swissprot
retrieve_blast.pl < YEL009C.blastp2.swissprot.temp.html
```

The NCBI blast server will respond with an ID number indicating the file in which the blast results are stored (with a line like ‘‘Obtained request ID: 940912366-18156-27559’’). That result file will then be

stored locally with a name like 940905064-15626-17267.txt, and can subsequently be read with Blast.pm or BPlite as described below.

Run the scripts run_blast_remote.pl, retrieve_blast.pl and blast_config.pl with the options ‘‘-h’’ or ‘‘-eg’’ for more examples on how to use Blast.pm to perform remote blasts.

III.4.3 Parsing BLAST reports with Blast.pm

No matter how Blast searches are run (locally or remotely, with or without a perl interface), they return large quantities of data that are tedious to sift through. Bioperl offers two different objects - Blast.pm and BPlite.pm (along with its minor modifications, BPpsilite and BPbl2seq) for parsing Blast reports.

The parser contained within the Blast.pm module is the original Blast parser developed for Bioperl. It is very full featured and has a large array of options and output formats. Typical syntax for parsing a blast report with Blast.pm is:

```
use Bio::Tools::Blast;
$blast = Bio::Tools::Blast->new(-file   =>'t/blast.report',
                                -signif => 1e-5,
                                -parse  => 1,
                                -stats  => 1,
                                -check_all_hits => 1, );
$blast->display();
$num_hits =  $blast->num_hits;
@hits  = $blast->hits;
$frac1 = $hits[1]->frac_identical;
@inds = $hits[1]->hsp->seq_inds('query', 'iden', 1);
```

Here the method ‘‘hits’’ returns an object containing the names of the sequences which produced a match and the ‘‘hsp’’ method returns a ‘‘high scoring pair’’ object containing the actual sequence alignments that each of the hits produced.

One very nice feature of the Blast.pm parser is being able to define an arbitrary ‘‘filter function’’ for use while parsing the Blast hits. With this feature, you can filter your results to just save hits with specific pattern in their id fields (eg ‘‘homo sapiens’’) or specific sequence patterns in a returned high-scoring-pair or just about anything else that can be found in the blast report record.

While the Blast object is parsing the report, each hit is checked by calling &filter($hit). All hits that generate false return values from `&filter` are screened out of the Blast object.. Note that the Blast object will normally stop parsing after the first non-significant hit or the first hit that does not pass the filter function. To force the Blast object to check all hits, include a ‘‘ -check_all_hits => 1’’ parameter. For example, to eliminate all hits with gaps or with less than 50% conserved residues one could use the following filter function:

```
sub filter { $hit=shift;
return ($hit->gaps == 0 and $hit->frac_conserved > 0.5); }
```

and use it like this:

```
$blastObj = Bio::Tools::Blast->new( '-file'      => '/tmp/blast.out',
                                    '-parse'     => 1,
                                    '-check_all_hits' => 1,
                                    '-filt_func' => \&filter );
```

Unfortunately the flexibility of the Blast.pm parser comes at a cost of complexity. As a result of this complexity and the fact that Blast.pm's original developer is no longer actively supporting the module, the Blast.pm parser has been difficult to maintain and has not been upgraded to handle the output of the newer blast options such as PSIBLAST and BL2SEQ. Consequently, the BPlite parser (described in the following section) is recommended for most blast parsing within bioperl.

III.4.4 Parsing BLAST reports with BPlite, BPpsilite and BPbl2seq

Because of the issues with Blast.pm discussed above, Ian Korf's BPlite parser has been recently ported to Bioperl. BPlite is less complex and easier to maintain than Blast.pm. Although it has fewer options and display modes than Blast.pm, you will probably find that it contains the functionality that you need. (One exception might be if you want to set up an arbitrary filter function - as described above - in which case you may want to use the Blast.pm parser.)

BPlite

The syntax for using BPlite is as follows where the method for retrieving hits is now called ''nextSbjct'' (for ''subject''), while the method for retrieving high-scoring-pairs is called ''nextHSP'':

```
use Bio::Tools::BPlite;
$report = new BPlite(-fh=>\*STDIN);
$report->query;
while(my $sbjct = $report->nextSbjct) {
     $sbjct->name;
     while (my $hsp = $sbjct->nextHSP) { $hsp->score; }
}
```

BPpsilite

BPpsilite and BPbl2seq are objects for parsing (multiple iteration) PSIBLAST reports and Blast bl2seq reports, respectively. They are both minor variations on the BPlite object.

The syntax for parsing a multiple iteration PSIBLAST report is as shown below. The only significant additions to BPlite are methods to determine the number of iterated blasts and to access the results from each iteration. The results from each iteration are parsed in the same manner as a (complete) BPlite object.

```
use Bio::Tools::BPpsilite;
$report = new BPpsilite(-fh=>\*STDIN);
$total_iterations = $report->number_of_iterations;
$last_iteration = $report->round($total_iterations)
while(my $sbjct =  $last_iteration ->nextSbjct) {
     $sbjct->name;
```

```
     while (my $hsp = $sbjct->nextHSP) {$hsp->score; }
}
```

BPbl2seq

BLAST bl2seq is a program for comparing and aligning two sequences using BLAST. Although the report format is similar to that of a conventional BLAST, there are a few differences. Consequently, the standard bioperl parsers Blast.pm and BPlite are unable to read bl2seq reports directly. From the user's perspective, the main difference between bl2seq and other blast reports is that the bl2seq report does not print out the name of the first of the two aligned sequences. Consequently, BPbl2seq has no way of identifying the name of one of the initial sequence unless it is explicitly passed to constructor as a second argument as in:

```
use Bio::Tools::BPbl2seq;
$report = Bio::Tools::BPbl2seq->new(-file => "t/bl2seq.out", -queryname => "ALEU_H
$matches = $report->match;
```

III.4.5 Parsing HMM reports (HMMER::Results)

Blast is not the only sequence-similarity-searching program supported by bioperl. HMMER is a Hidden Markov-chain Model (HMM) program that (among other capabilities) enables sequence similarity searching. Bioperl does not currently provide a perl interface for running HMMER. However, bioperl does provide a HMMER report parser with the (perhaps not too descriptive) name of Results.

Results can parse reports generated both by the HMMER program hmmsearch - which searches a sequence database for sequences similar to those generated by a given HMM - and the program hmmpfam - which searches a HMM database for HMMs which match domains of a given sequence. For hmmsearch, a series of HMMER::Set objects are made, one for each sequence. For hmmpfam searches, only one Set object is made. Sample usage for parsing a hmmsearch report might be:

```
use Bio::Tools::HMMER::Results;
$res = new Bio::Tools::HMMER::Results('-file' => 'output.hmm' ,
                                      '-type' => 'hmmsearch');
foreach $seq ( $res->each_Set ) {
    print "Sequence bit score is ", $seq->bits, "\n";
    foreach $domain ( $seq->each_Domain ) {
        print " Domain start ", $domain->start, " end ",
            $domain->end," score ",$domain->bits,"\n";
    }
}
```

III.5 Creating and manipulating sequence alignments

Once one has identified a set of similar sequences, one often needs to create an alignment of those sequences. Bioperl offers several perl objects to facilitate sequence alignment: pSW, Clustalw.pm, TCoffee.pm and the bl2seq option of StandAloneBlast. All of these objects take as arguments a reference to an array of (unaligned) Seq objects. All (except bl2seq) return a reference to a SimpleAlign

object. bl2seq can also produce a SimpleAlign object when it is combined with AlignIO (see below section III.5.2).

III.5.1 Aligning 2 sequences with Smith-Waterman (pSW)

The Smith-Waterman (SW) algorithm is the standard method for producing an optimal alignment of two sequences. Bioperl supports the computation of SW alignments via the pSW object. The SW algorithm itself is implemented in C and incorporated into bioperl using an XS extension. This has significant efficiency advantages but means that pSW will **not** work unless you have compiled the bioperl-ext package. If you have compiled the bioperl-ext package, usage is simple, where the method align_and_show displays the alignment while pairwise_alignment produces a (reference to) a SimpleAlign object.

```
use Bio::Tools::pSW;
$factory = new Bio::Tools::pSW( '-matrix' => 'blosum62.bla',
                                '-gap' => 12,
                                '-ext' => 2, );
$factory->align_and_show($seq1, $seq2, STDOUT);
$aln = $factory->pairwise_alignment($seq1, $seq2);
```

SW matrix, gap and extension parameters can be adjusted as shown. Bioperl comes standard with blosum62 and gonnet250 matrices. Others can be added by the user. For additional information on accessing the SW algorithm via pSW see the example script pSW.pl and the documentation in pSW.pm.

III.5.2 Aligning 2 sequences with Blast using bl2seq and AlignIO

As an alternative to Smith-Waterman, two sequences can also be aligned in Bioperl using the bl2seq option of Blast within the StandAloneBlast object. To get an alignment - in the form of a SimpleAlign object - using bl2seq, you need to parse the bl2seq report with the AlignIO file format reader as follows:

```
$factory = Bio::Tools::StandAloneBlast->new('outfile' => 'bl2seq.out');
$bl2seq_report = $factory->bl2seq($seq1, $seq2);
# Use AlignIO.pm to create a SimpleAlign object from the bl2seq report
$str = Bio::AlignIO->new('-file '=>' bl2seq.out',
                         '-format' => 'bl2seq');
$aln = $str->next_aln();
```

III.5.3 Aligning multiple sequences (Clustalw.pm, TCoffee.pm)

For aligning multiple sequences (ie two or more), bioperl offers a perl interface to the bioinformatics-standard clustalw and tcoffee programs. Clustalw has been a leading program in global multiple sequence alignment (MSA) for several years. TCoffee is a relatively recent program - derived from clustalw - which has been shown to produce better results for local MSA.

To use these capabilities, the clustalw and/or tcoffee programs themselves need to be installed on the host system. In addition, the environmental variables CLUSTALDIR and TCOFFEEDIR need to be set to the directories containg the executables. See section I.3 and the Clustalw.pm and TCoffee.pm module documentation for information on downloading and installing these programs.

From the user's perspective, the bioperl syntax for calling Clustalw.pm or TCoffee.pm is almost identical. The only differences are the names of the modules themselves appearing in the initial ''use'' and constructor statements and the names of the some of the individual program options and parameters.

In either case, initially, a ''factory object'' must be created. The factory may be passed most of the parameters or switches of the relevant program. In addition, alignment parameters can be changed and/or examined after the factory has been created. Any parameters not explicitly set will remain as the underlying program's defaults. Clustalw.pm/TCoffee.pm output is returned in the form of a SimpleAlign object. It should be noted that some Clustalw and TCoffee parameters and features (such as those corresponding to tree production) have not been implemented yet in the Perl interface.

Once the factory has been created and the appropriate parameters set, one can call the method `align()` to align a set of unaligned sequences, or `profile_align()` to add one or more sequences or a second alignment to an initial alignment. Input to `align()` consists of a set of unaligned sequences in the form of the name of file containing the sequences or a reference to an array of Bio::Seq objects. Typical syntax is shown below. (We illustrate with Clustalw.pm, but the same syntax - except for the module name - would work for TCoffee.pm)

```
use Bio::Tools::Run::Alignment::Clustalw;
@params = ('ktuple' => 2, 'matrix' => 'BLOSUM');
$factory = Bio::Tools::Run::Alignment::Clustalw->new(@params);
$ktuple = 3;
$factory->ktuple($ktuple);  # change the parameter before executing
$seq_array_ref = \@seq_array;
    # where @seq_array is an array of Bio::Seq objects
$aln = $factory->align($seq_array_ref);
```

Clustalw.pm/TCoffee.pm can also align two (sub)alignments to each other or add a sequence to a previously created alignment by using the profile_align method. For further details on the required syntax and options for the profile_align method, the user is referred to the Clustalw.pm/TCoffee.pm documentation. The user is also encouraged to run the script clustalw.pl in the examples directory.

III.5.4 Manipulating / displaying alignments (SimpleAlign, UnivAln)

As described in section II.2, bioperl currently includes two alignment objects, SimpleAlign and UnivAln. SimpleAlign objects are usually more useful, since they are directly produced by bioperl alignment creation objects (eg Clustalw.pm and pSW) and can be used to read and write multiple alignment formats via AlignIO.

However, SimpleAlign currently only offers limited functionality for alignment manipulation. One useful method offered by SimpleAlign is `consensus_string()`. This method returns a string with the

most common residue in the alignment at each string position. An optional threshold ranging from 0 to 100 can be passed to consensus_string. If the consensus residue appears in fewer than the threshold % of the sequences, consensus_string will return a ‘‘?’’ at that location. Typical usage is:

```
  use Bio::SimpleAlign;
  $aln = Bio::SimpleAlign->new('t/alnfile.fasta');
  $threshold_percent = 60;
  $str = $aln->consensus_string($threshold_percent)
```

UnivAln, on the other hand, offers a variety of methods for ‘‘slicing and dicing’’ an alignment including methods for removing gaps, reverse complementing specified rows and/or columns of an alignment, and extracting consensus sequences with specified thresholds for the entire alignment or a sub-alignment. Typical usage is:

```
  use Bio::UnivAln;
  $aln = Bio::UnivAln->new('t/alnfile.fasta');
  $resSlice1 = $aln->remove_gaps(); # original sequences without gaps
  $resSlice2 = $aln->revcom([1,3]); # reverse complement, rows 1+3 only
  $resSlice3 = $aln->consensus(0.6, [1,3]);
       # 60% majority, columns 1+3 only
```

Many additional - and more intricate - methods exist. See the UnivAln documentation. Note that if you do want to use UnivAln’s methods on an alignment, you will first need to convert the alignment into fasta format (which can be done via the SimpleAlign and AlignIO objects discussed above.)

III.6 Searching for genes and other structures on genomic DNA (Genscan, Sim4, ESTScan, MZEF)

Automated searching for putative genes, coding sequences and other functional units in genomic and expressed sequence tag (EST) data has become very important as the available quantity of sequence data has rapidly increased. Many gene searching programs currently exist. Each produces reports containing predictions that must be read manually or parsed by automated report readers.

Parsers for four widely used gene prediction programs- Genscan, Sim4, ESTScan and MZEF - are currently available or under active development in bioperl. The interfaces for the four parsers are similar. We illustrate the usage for Genscan and Sim4 here. The syntax is relatively self-explanatory; further details are available in the module documentation in the Bio::Tools directory.

```
  use Bio::Tools::Genscan;
  $genscan = Bio::Tools::Genscan->new(-file => 'result.genscan');
# $gene is an instance of Bio::Tools::Prediction::Gene
# $gene->exons() returns an array of Bio::Tools::Prediction::Exon objects
  while($gene = $genscan->next_prediction())
      { @exon_arr = $gene->exons(); }
  $genscan->close();

  use Bio::Tools::Sim4::Results;
   $sim4 = new Bio::Tools::Sim4::Results(-file=> 't/sim4.rev', -estisfirst=>0);
 # $exonset is-a Bio::SeqFeature::Generic with Bio::Tools::Sim4::Exons
 # as sub features
```

```
   $exonset = $sim4->next_exonset;
   @exons = $exonset->sub_SeqFeature();
# $exon is-a Bio::SeqFeature::FeaturePair
   $exon = 1;
   $exonstart = $exons[$exon]->start();
   $estname = $exons[$exon]->est_hit()->seqname();
  $sim4->close();
```

III.7 Developing machine readable sequence annotations

Historically, annotations for sequence data have been entered and read manually in flat-file or relational databases with relatively little concern for machine readability. More recent projects - such as EBI's Ensembl project and the efforts to develop an XML molecular biology data specification - have begun to address this limitation. Because of its strengths in text processing and regular-expression handling, perl is a natural choice for the computer language to be used for this task. And bioperl offers numerous tools to facilitate this process - several of which are described in the following sub-sections.

III.7.1 Representing sequence annotations (Annotation, SeqFeature)

As of the 0.7 release of bioperl, the fundamental sequence object, Seq, can have multiple sequence feature (SeqFeature) objects - eg Gene, Exon, Promoter objects - associated with it. A Seq object can also have an Annotation object (used to store database links, literature references and comments) associated with it. Creating a new SeqFeature and Annotation and associating it with a Seq is accomplished with syntax like:

```
  $feat = new Bio::SeqFeature::Generic('-start' => 40,
                                       '-end' => 80,
                                       '-strand' => 1,
                                       '-primary' => 'exon',
                                       '-source' => 'internal' );
  $seqobj->add_SeqFeature($feat); # Add the SeqFeature to the parent
  $seqobj->annotation(new Bio::Annotation
      ('-description' => 'desc-here'));
```

Once the features and annotations have been associated with the Seq, they can be with retrieved, eg:

```
  @topfeatures = $seqobj->top_SeqFeatures(); # just top level, or
  @allfeatures = $seqobj->all_SeqFeatures(); # descend into sub features
  $ann = $seqobj->annotation(); # annotation object
```

The individual components of a SeqFeature can also be set or retrieved with methods including:

```
  # attributes which return numbers
  $feat->start          # start position
  $feat->end            # end position

  $feat->strand         # 1 means forward, -1 reverse, 0 not relevant
```

```
# attributes which return strings
$feat->primary_tag    # the main 'name' of the sequence feature,
                      # eg, 'exon'
$feat->source_tag     # where the feature comes from, eg'BLAST'

# attributes which return Bio::PrimarySeq objects
$feat->seq            # the sequence between start,end
$feat->entire_seq     # the entire sequence

# other useful methods include
$feat->overlap($other)  # do SeqFeature $feat and SeqFeature $other overlap?
$feat->contains($other) # is $other completely within $feat?
$feat->equals($other)   # do $feat and $other completely $agree?
$feat->sub_SeqFeatures  # create/access an array of subsequence features
```

III.7.2 Representing and large and/or changing sequences (LiveSeq,LargeSeq)

Very large sequences and/or data files with sequences that are frequently being updated present special problems to automated sequence-annotation storage and retrieval projects. Bioperl's LargeSeq and LiveSeq objects are designed to address these two situations.

LargeSeq

A LargeSeq object is a SeqI compliant object that stores a sequence as a series of files in a temporary directory (see sect II.1 for a definition of SeqI objects). The aim is to enable storing very large sequences (eg, > 100MBases) without running out of memory and, at the same time, preserving the familiar bioperl Seq object interface. As a result, from the users perspective, using a LargeSeq object is almost identical to using a Seq object. The principal difference is in the format used in the SeqIO calls. Another difference is that the user must remember to only read in small chunks of the sequence at one time. These differences are illustrated in the following code:

```
$seqio = new Bio::SeqIO('-format'=>'largefasta',
                        '-file'  =>'t/genomic-seq.fasta');
$pseq = $seqio->next_seq();
$plength = $pseq->length();
$last_4 = $pseq->subseq($plength-3,$plength);  # this is OK

#On the other hand, the next statement would
#probably cause the machine to run out of memory
#$lots_of_data = $pseq->seq();  #NOT OK for a large LargeSeq object
```

LiveSeq

The LiveSeq object addresses the need for a sequence object capable of handling sequence data that may be changing over time. In such a sequence, the precise locations of features along the sequence may change. LiveSeq deals with this issue by re-implementing the sequence object internally as a ''double linked chain.'' Each element of the chain is connected to other two elements (the PREVious and the NEXT one). There is no absolute position (like in an array), hence if positions are important, they need to be computed (methods are provided). Otherwise it's easy to keep track of the elements with their

‘‘LABELs’’. There is one LABEL (think of it as a pointer) to each ELEMENT. The labels won’t change after insertions or deletions of the chain. So it’s always possible to retrieve an element even if the chain has been modified by successive insertions or deletions.

Although the implementation of the LiveSeq object is novel, its bioperl user interface is unchanged since LiveSeq implements a PrimarySeqI interface (recall PrimarySeq is the subset of Seq without annotations or SeqFeatures - see sec II.1). Consequently syntax for using LiveSeq objects is familiar although a modified version of SeqIO called LiveSeq::IO::Bioperl needs to be used to actually load the data, eg:

```
$loader=Bio::LiveSeq::IO::BioPerl->load('-db'=>"EMBL",
                                        '-file'=>"t/factor7.embl");
$gene=$loader->gene2liveseq('-gene_name' => "factor7");
$id = $gene->get_DNA->display_id ;
$maxstart = $gene->maxtranscript->start;
```

Creating, maintaining and querying of LiveSeq genes is quite memory and processor intensive. Consequently, any additional information relating to mutational changes in a gene need to be stored separately from the sequence data itself. The next section describes the mutation and polymorphism objects used to accomplish this.

III.7.3 Representing related sequences - mutations, polymorphisms etc (Allele, SeqDiff,)

Bio::LiveSeq::Mutation object allows for a basic description of a sequence change in DNA or cDNA sequence of a gene. Bio::LiveSeq::Mutator takes in mutations, applies them to a LiveSeq gene and returns a set of Bio::Variation objects describing the net effect of the mutation on the gene at the DNA, RNA and protein stages.

The objects in Bio::Variation and Bio::LiveSeq directory were originally designed for the ‘‘Computational Mutation Expression Toolkit’’ project at European Bioinformatics Institute (EBI). The result of using them to mutate a gene is a holder object, ’SeqDiff’, that can be printed out or queried for specific information. For example, to find out if restriction enzyme changes caused by a mutation are exactly the same in DNA and RNA sequences, we can write:

```
use Bio::LiveSeq::IO::BioPerl;
use Bio::LiveSeq::Mutator;
use Bio::LiveSeq::Mutation;

$loader=Bio::LiveSeq::IO::BioPerl->load('-file' => "$filename");
$gene=$loader->gene2liveseq('-gene_name' => $gene_name);
$mutation = new Bio::LiveSeq::Mutation ('-seq' =>'G',
                                        '-pos' => 100,
                                       );
$mutate = Bio::LiveSeq::Mutator->new('-gene' => $gene,
                                     '-numbering' => "coding"
                                    );
$mutate->add_Mutation($mutation);
$seqdiff = $mutate->change_gene();
$DNA_re_changes = $seqdiff->DNAMutation->restriction_changes;
$RNA_re_changes = $seqdiff->RNAChange->restriction_changes;
```

```
$DNA_re_changes eq $RNA_re_changes or print "Different!\n";
```

For a complete working script, see the change_gene.pl script in the examples directory. For more details on the use of these objects see the documentation in the modules as well as the original documentation for the ‘‘Computational Mutation Expression Toolkit’’ project at http://www.ebi.ac.uk/mutations/toolkit/.

III.7.4 Sequence XML representations - generation and parsing (SeqIO::game)

The previous subsections have described tools for automated sequence annotation by the creation of an ‘‘object layer’’ on top of a traditional database structure. XML takes a somewhat different approach. In XML, the data structure is unmodified, but machine readability is facilitated by using a data-record syntax with special flags and controlled vocabulary.

Bioperl supports a set of XML flags and vocabulary words for molecular biology - called bioxml - detailed at http://www.bioxml.org/dtds/current/ The idea is that any bioxml features can be turned into bioperl Bio::Seq annotations. Conversely Seq object features and annotations can be converted to XML so that they become available to any other systems that are XML (and bioxml) compliant. Typical usage is shown below. No special syntax is required by the user. Note that some Seq annotation will be lost when using bioxml in this manner - since in its current implementation, bioxml does not support all the annotation information available in Seq objects.

```
$str = Bio::SeqIO->new('-file'=> 't/test.game',
                       '-format' => 'game');
$seq = $str->next_primary_seq();
$id = $seq->id;
@feats = $seq->all_SeqFeatures();
$first_primary_tag = $feats[0]->primary_tag;
```

IV. Related projects - biocorba, biopython, biojava, Ensembl, AnnotationWorkbench / bioperl-gui

There are several ‘‘sister projects’’ to bioperl currently under development. These include biocorba, biopython, biojava, Ensembl, and the Annotation Workbench (which includes Bioperl-gui). These are all large complex projects and describing them in detail here will not be attempted. However a brief introduction seems appropriate since, in the future, they may each provide significant added utility to the bioperl user.

IV.1 Biocorba

Interface objects have facilitated interoperability between bioperl and other perl packages such as Ensembl and the Annotation Workbench. However, interoperability between bioperl and packages written in other languages requires additional support software. CORBA is one such framework for interlanguage support, and the biocorba project is currently implementing a CORBA interface for bioperl. With biocorba, objects written within bioperl will be able to communicate with objects written in biopython and biojava (see the next subsection). For more information, se the biocorba project website at http://biocorba.org/.

IV.2 Biopython and biojava

Biopython and biojava are open source projects with very similar goals to bioperl. However their code is implemented in python and java, respectively. With the development of interface objects and biocorba, it is possible to write java or python objects which can be accessed by a bioperl script. Or to call bioperl objects from java or python code. Since biopython and biojava are more recent projects than bioperl, most effort to date has been to port bioperl functionality to biopython and biojava rather than the other way around. However, in the future, some bioinformatics tasks may prove to be more effectively implemented in java or python in which case being able to call them from within bioperl will become more important. For more information, go to the biojava http://biojava.org/ and biopython http://biopython.org/ websites.

IV.3 Ensembl

Ensembl is an ambitious automated-genome-annotation project at EBI. Much of Ensembl's code is based on bioperl and Ensembl developers, in turn, have contributed significant pieces of code to bioperl. In particular, the bioperl code for automated sequence annotation has been largely contributed by Ensembl developers. (The close association between bioperl and Ensembl is not surprising, since the same individual - Ewan Birney - has been coordinating both projects). Describing Ensembl and its capabilities is far beyond the scope of this tutorial The interested reader is referred to the Ensembl website at http://www.ensembl.org/ .

IV.4 The Annotation Workbench and bioperl-gui

The Annotation Workbench (AW) being developed at the Plant Biotechnology Institute of the National Research Council of Canada is designed to be an integrated suite of tools for examining a sequence, predicting gene structure, and creating annotations. The workbench features a graphical user interface and is implemented completely in perl. Information about the AW is available at http://bioinfo.pbi.nrc.ca/dblock/wiki/html/Bioinfo/AnnotationWorkbench.htm A goal of the AW team is to port much of the functionality of the AW to bioperl. The porting process has begun and displaying a

Seq object graphically is now possible. You can download the current version of the gui software from the bioperl bioperl-gui CVS directory at
http://cvs.bioperl.org/cgi-bin/viewcvs/viewcvs.cgi/bioperl-gui/?cvsroot=bioperl

---

V.1 Appendix: Public Methods of Bioperl Objects

Appendix V.1 lists the public methods for the principal bioperl objects. The methods are grouped together under the name of the ‘‘ancestor’’ object from which the object inherits the method. Knowing the name of the ancestor object is useful since the ancestor module will contain the documentation describing the method.

Since nearly all bioperl objects inherit from Bio::Root::RootI, the methods inherited from Bio::Root::RootI are listed separately - and only once.

If a listing of public methods for a bioperl object not listed in this appendix is required, the listing can be obtained by running the bptutorial.pl script using with command number 100 as in: perl -w bptutorial.pl 100 Bio::Tools::SeqStats (where Bio::Tools::SeqStats would be replaced with the name of the bioperl object for which the methods list is needed - see Appendix V.2 for details)

```
***Methods for Object Bio::Root::RootI ********
```

Methods taken from package Bio::Root::RootI DESTROY gensym new qualify qualify_to_ref stack_trace stack_trace_dump tempdir tempfile throw ungensym verbose warn

```
***Methods for Object Bio::AlignIO ********

Methods taken from package Bio::AlignIO
PRINT   READLINE   TIEHANDLE   close   fh   newFh
next_aln   write_aln

***Methods for Object Bio::DB::GenBank ********

Methods taken from package Bio::DB::NCBIHelper
get_Stream_by_batch   get_params

Methods taken from package Bio::DB::RandomAccessI
get_Seq_by_acc   get_Seq_by_id

Methods taken from package Bio::DB::WebDBSeqI
GET   HEAD   POST   PUT   default_format   get_Stream_by_acc
get_Stream_by_id   get_request   get_seq_stream   postprocess_data   proxy   reques
retrieval_type   ua   url_base_address   url_params

***Methods for Object Bio::Index::Fasta ********

Methods taken from package Bio::DB::RandomAccessI
get_Seq_by_acc   get_Seq_by_id

Methods taken from package Bio::DB::SeqI
get_PrimarySeq_stream   get_Seq_by_primary_id   get_all_primary_ids

Methods taken from package Bio::Index::Abstract
```

```
O_CREAT   O_RDWR   add_record   db   dbm_package   filename
get_stream   make_index   open_dbm   pack_record   unpack_record   write_flag

Methods taken from package Bio::Index::AbstractSeq
fetch

Methods taken from package Bio::Index::Fasta
default_id_parser   id_parser

***Methods for Object Bio::LocatableSeq ********

Methods taken from package Bio::LocatableSeq
get_nse

Methods taken from package Bio::PrimarySeqI
GCG_checksum   accession_number   ary   can_call_new   desc   display_id
getseq   id   moltype   out_fasta   primary_id   revcom
seq   seq_len   setseq   str   subseq   translate
translate_old   trunc   type

Methods taken from package Bio::RangeI
carp   confess   contains   croak   end   equals
intersection   length   new   overlap_extent   overlaps   start
strand   union

Methods taken from package Bio::Seq
accession   add_SeqFeature   add_date   add_secondary_accession   annotation   divi
each_date   each_secondary_accession   feature_count   keywords   molecule   primar
species   sv

Methods taken from package Bio::SeqI
all_SeqFeatures   top_SeqFeatures   write_GFF

***Methods for Object Bio::Seq ********

Methods taken from package Bio::PrimarySeqI
GCG_checksum   accession_number   ary   can_call_new   carp   confess
croak   desc   display_id   getseq   id   length
moltype   out_fasta   primary_id   revcom   seq   seq_len
setseq   str   subseq   translate   translate_old   trunc
type

Methods taken from package Bio::Seq
accession   add_SeqFeature   add_date   add_secondary_accession   annotation   divi
each_date   each_secondary_accession   feature_count   keywords   molecule   primar
species   sv

Methods taken from package Bio::SeqI
all_SeqFeatures   top_SeqFeatures   write_GFF

***Methods for Object Bio::SeqIO ********

Methods taken from package Bio::SeqIO
PRINT   READLINE   TIEHANDLE   close   fh   moltype
newFh   next_primary_seq   next_seq   write_seq

***Methods for Object Bio::SimpleAlign ********

Methods taken from package Bio::SimpleAlign
addSeq   alpha_startend   column_from_residue_number   consensus_aa   consensus_str
eachSeqWithId   each_alphabetically   get_displayname   id   is_flush   length_aln
```

map_chars maxdisplayname_length maxname_length maxnse_length no_residues
percentage_identity purge read_MSF read_Pfam read_Pfam_file read_Prodom
read_fasta read_mase read_selex read_stockholm removeSeq set_displayname
set_displayname_count set_displayname_flat set_displayname_normal sort_alphab
write_Pfam write_clustalw write_fasta write_selex

Methods for Object Bio::Tools::BPbl2seq *****

Methods taken from package Bio::RangeI
carp confess contains croak end equals
intersection length new overlap_extent overlaps start
strand union

Methods taken from package Bio::SeqFeature::FeaturePair
feature1 feature2 hend hframe hprimary_tag hscore
hseqname hsource_tag hstart hstrand invert

Methods taken from package Bio::SeqFeature::Generic
add_sub_SeqFeature add_tag_value annotation attach_seq entire_seq flush_s
frame remove_tag score seq seqname slurp_gff_file

Methods taken from package Bio::SeqFeature::SimilarityPair
bits from_searchResult query significance subject

Methods taken from package Bio::SeqFeatureI
all_tags each_tag_value gff2_string gff_string has_tag primary_tag
source_tag sub_SeqFeature

Methods taken from package Bio::Tools::BPbl2seq
P homologySeq hs match percent positive
qs querySeq sbjctSeq ss

Methods for Object Bio::Tools::BPlite *****

Methods taken from package Bio::Tools::BPlite
database fh nextSbjct pattern qlength query
query_pattern_location

Methods for Object Bio::Tools::BPpsilite *****

Methods taken from package Bio::Tools::BPpsilite
database fh number_of_iterations pattern qlength query
query_pattern_location round

Methods for Object Bio::Tools::Blast *****

Methods taken from package Bio::Root::Object
clear_err clone compress_file containment debug delete_file
destroy display dont_warn err err_state err_string
fatal_on_warn fh file file_date find_object has_name
has_warning index make monitor name parent
print_err read record_err set_display set_err_data set_log_err
set_read set_stats show src_obj strict strictness
terse testing to_string uncompress_file verbosity warn_on_fatal
xref

Methods taken from package Bio::Tools::Blast
ambiguous_aln carp confess croak db_local db_remote
expect filter gap_creation gap_extension gapped highest_expect
highest_p highest_signif hit hits homol_data is_signif
karlin_altschul lowest_expect lowest_p lowest_signif matrix min_length

```
num_hits   overlap   s   signif   signif_fmt   table
table_labels   table_labels_tiled   table_tiled   to_html   word_size

Methods taken from package Bio::Tools::SeqAnal
best   database   database_letters   database_release   database_seqs   date
length   parse   program   program_version   query   query_desc
roman2int   run   set_date

Methods taken from package Exporter
export   export_fail   export_ok_tags   export_tags   export_to_level   import
require_version

***Methods for Object Bio::Tools::CodonTable ********

Methods taken from package Bio::Root::RootI
DESTROY   gensym   new   qualify   qualify_to_ref   stack_trace
stack_trace_dump   tempdir   tempfile   throw   ungensym   verbose
warn

Methods taken from package Bio::Tools::CodonTable
id   is_start_codon   is_ter_codon   is_unknown_codon   name   revtranslate
translate   translate_strict

***Methods for Object Bio::Tools::Genscan ********

Methods taken from package Bio::SeqAnalysisParserI
carp   confess   croak   next_feature   parse

Methods taken from package Bio::Tools::AnalysisResult
analysis_date   analysis_method   analysis_method_version   analysis_query   analys

Methods taken from package Bio::Tools::Genscan
next_prediction

***Methods for Object Bio::Tools::HMMER::Results ********

Methods taken from package Bio::Tools::HMMER::Results
add_Domain   add_Set   carp   confess   croak   dictate_hmm_acc
domain_bits_cutoff_from_evalue   each_Domain   each_Set   filter_on_cutoff   get_Se
highest_noise   lowest_true   number   write_FT_output   write_GDF   write_GDF_bits
write_ascii_out   write_scores_bits

***Methods for Object Bio::Tools::OddCodes ********

Methods taken from package Bio::Tools::OddCodes
Dayhoff   Sneath   Stanfel   charge   chemical   functional
hydrophobic   structural

***Methods for Object Bio::Tools::RestrictionEnzyme ********

Methods taken from package Bio::Tools::RestrictionEnzyme
annotate_seq   available   available_list   cut_locations   cut_seq   cuts_after
is_available   name   palindromic   revcom   seq   site
string

Methods taken from package Exporter
export   export_fail   export_ok_tags   export_tags   export_to_level   import
require_version

***Methods for Object Bio::Tools::Run::Alignment::Clustalw ********
```

```
Methods taken from package Bio::Tools::Run::Alignment::Clustalw
AUTOLOAD   align   exists_clustal   profile_align

***Methods for Object Bio::Tools::Run::Alignment::TCoffee ********

Methods taken from package Bio::Tools::Run::Alignment::TCoffee
AUTOLOAD   align   exists_tcoffee   profile_align

***Methods for Object Bio::Tools::Run::StandAloneBlast ********

Methods taken from package Bio::Tools::Run::StandAloneBlast
AUTOLOAD   bl2seq   blastall   blastpgp   exists_blast

***Methods for Object Bio::Tools::SeqPattern ********

Methods taken from package Bio::Root::Object
clear_err   clone   compress_file   containment   debug   delete_file
destroy   display   dont_warn   err   err_state   err_string
fatal_on_warn   fh   file   file_date   find_object   has_name
has_warning   index   make   monitor   name   parent
print_err   read   record_err   set_display   set_err_data   set_log_err
set_read   set_stats   show   src_obj   strict   strictness
terse   testing   to_string   uncompress_file   verbosity   warn_on_fatal
xref

Methods taken from package Bio::Tools::SeqPattern
alphabet_ok   expand   revcom   str   type

***Methods for Object Bio::Tools::SeqStats ********

Methods taken from package Bio::Tools::SeqStats
count_codons   count_monomers   get_mol_wt

***Methods for Object Bio::Tools::SeqWords ********

Methods taken from package Bio::Root::Object
clear_err   clone   compress_file   containment   debug   delete_file
destroy   display   dont_warn   err   err_state   err_string
fatal_on_warn   fh   file   file_date   find_object   has_name
has_warning   index   make   monitor   name   parent
print_err   read   record_err   set_display   set_err_data   set_log_err
set_read   set_stats   show   src_obj   strict   strictness
terse   testing   to_string   uncompress_file   verbosity   warn_on_fatal
xref

Methods taken from package Bio::Tools::SeqWords
count_words

***Methods for Object Bio::Tools::Sigcleave ********

Methods taken from package Bio::PrimarySeqI
GCG_checksum   accession_number   ary   can_call_new   carp   confess
croak   desc   display_id   getseq   id   length
moltype   out_fasta   primary_id   revcom   seq   seq_len
setseq   str   subseq   translate   translate_old   trunc
type

Methods taken from package Bio::Seq
accession   add_SeqFeature   add_date   add_secondary_accession   annotation   divi
each_date   each_secondary_accession   feature_count   keywords   molecule   primar
species   sv
```

```
Methods taken from package Bio::SeqI
all_SeqFeatures   top_SeqFeatures   write_GFF

Methods taken from package Bio::Tools::Sigcleave
debug   dont_warn   fatal_on_warn   monitor   pretty_print   signals
strictness   testing   threshold   verbosity   warn_on_fatal

***Methods for Object Bio::Tools::Sim4::Results ********

Methods taken from package Bio::SeqAnalysisParserI
carp   confess   croak   next_feature   parse

Methods taken from package Bio::Tools::AnalysisResult
analysis_date   analysis_method   analysis_method_version   analysis_query   analys

Methods taken from package Bio::Tools::Sim4::Results
basename   dirname   fileparse   fileparse_set_fstype   next_exonset   parse_next_a

***Methods for Object Bio::Tools::pSW ********

Methods taken from package Bio::Tools::AlignFactory
kbyte   report   set_memory_and_report

Methods taken from package Bio::Tools::pSW
align_and_show   ext   gap   matrix   pairwise_alignment

***Methods for Object Bio::UnivAln ********

Methods taken from package Bio::UnivAln
abort   access   acos   aln   alphabet_check   asctime
asin   atan   basename   carp   catch   ceil
clock   col_descs   col_ids   complement   confess   consensus
copy   cosh   croak   ctermid   ctime   cuserid
desc   descffmt   difftime   dirname   dup   dup2
equal_nogaps   equalize_lengths   ffmt   fileparse   fileparse_set_fstype   floor
fmod   fpathconf   frexp   gap_free_cols   gap_free_inds   gap_free_sites
height   id   inplace   invar_inds   invar_sites   isalnum
isalpha   iscntrl   isdigit   isgraph   islower   isprint
ispunct   isspace   isupper   isxdigit   layout   ldexp
localeconv   log10   lseek   map_c   map_r   mblen
mbstowcs   mbtowc   mkfifo   mktime   modf   names
new   no_allgap_inds   no_allgap_sites   numbering   out_bad   out_fasta
out_fasta2   out_raw   out_raw2   out_readseq   parse   parse_bad
parse_fasta   parse_raw   parse_unknown   pathconf   pause   remove_gaps
revcom   reverse   row_descs   row_ids   samelength   seqs
setlocale   setpgid   setsid   sigaction   sigpending   sigprocmask
sigsuspend   sinh   special_free_inds   special_free_sites   strcoll   strftime
strtod   strtol   strtoul   strxfrm   sysconf   tan
tanh   tcdrain   tcflow   tcflush   tcgetpgrp   tcsendbreak
tcsetpgrp   tmpnam   ttyname   type   tzname   tzset
uname   unknown_free_inds   unknown_free_sites   var_inds   var_sites   wcstombs
wctomb   width

Methods taken from package Exporter
export   export_fail   export_ok_tags   export_tags   export_to_level   import
require_version

***Methods for Object Bio::Variation::Allele ********

Methods taken from package Bio::DBLinkContainerI
carp   confess   croak   each_DBLink
```

```
Methods taken from package Bio::PrimarySeqI
GCG_checksum   accession_number   ary   can_call_new   desc   display_id
getseq   id   length   moltype   out_fasta   primary_id
revcom   seq   seq_len   setseq   str   subseq
translate   translate_old   trunc   type

Methods taken from package Bio::Variation::Allele
add_DBLink   is_reference   repeat_count   repeat_unit

***Methods for Object Bio::Variation::DNAMutation ********

Methods taken from package Bio::DBLinkContainerI
carp   confess   croak   each_DBLink

Methods taken from package Bio::RangeI
contains   end   equals   intersection   length   new
overlap_extent   overlaps   start   strand   union

Methods taken from package Bio::SeqFeature::Generic
add_sub_SeqFeature   add_tag_value   annotation   attach_seq   entire_seq   flush_s
frame   remove_tag   score   seq   seqname   slurp_gff_file

Methods taken from package Bio::SeqFeatureI
all_tags   each_tag_value   gff2_string   gff_string   has_tag   primary_tag
source_tag   sub_SeqFeature

Methods taken from package Bio::Variation::DNAMutation
CpG   RNAChange   sysname

Methods taken from package Bio::Variation::VariantI
SeqDiff   add_Allele   add_DBLink   allele_mut   allele_ori   dnStreamSeq
each_Allele   id   isMutation   label   mut_number   numbering
proof   region   region_value   restriction_changes   status   upStreamSeq

***Methods for Object Bio::Variation::SeqDiff ********

Methods taken from package Bio::Variation::SeqDiff
aa_mut   aa_ori   add_Gene   add_Variant   alignment   cds_end
cds_start   chromosome   description   dna_mut   dna_ori   each_Gene
each_Variant   gene_symbol   id   moltype   numbering   offset
rna_id   rna_mut   rna_offset   rna_ori   seqobj   sysname
trivname
```

V.2 Appendix: Tutorial demo scripts

The following scripts demonstrate many of the features of bioperl. To run all the demos run:

```
> perl -w  bptutorial.pl 0
```

To run a subset of the scripts do

```
> perl -w  bptutorial.pl
```

and use the displayed help screen.