Table of Contents

Chapter 1. BioPerlTutorial - a tutorial for bioperl

BioPerlTutorial - a tutorial for bioperl This document is available as HTML at http://www.bioperl.org/Core/bptutorial.html and as 'bptutorial.pl' in the Bioperl distribution.

AUTHOR

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



Copyright Peter Schattner



Contributions, additions and corrections have been made
to this document by the following individuals:



Jason Stajich
Heikki Lehvaslaiho
Brian Osborne
Hilmar Lapp
Chris Dagdigian
```

DESCRIPTION

```
 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

Bioperl is a collection of perl modules that facilitate the development of perl scripts for bioinformatics applications. As such, it does not include ready to use programs in the sense that many 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 bioinformatics 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.
- The POD documentation should contain runnable code in the SYNOPSIS section which is meant to illustrate the use of a module and its methods.

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. A very useful interface for finding one's way within all the module documentation can be found at http://doc.bioperl.org/bioperl-live/. This interface lists all bioperl modules and descriptions of all of their methods.

One potential problem in locating the correct documentation is that multiple methods in different modules may all share the same name. Moreover, because of perl's complex method of "inheritance", it is not often clear which of the identically named methods is being called by a given object. One way to resolve this question is by using the software described in Appendix V.1.

For those who prefer more visual descriptions, http://bioperl.org/Core/Latest/modules.html also offers links to three PDF files which contain schematics that describe how many of the bioperl objects related to one another.

In addition, a bioperl online course is available on the web at http://www.pasteur.fr/recherche/unites/sis/formation/bioperl. The user is also referred to numerous bioperl scripts in the scripts/ and examples/ directories (see bioperl.pod for a description of all these scripts).

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 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, from http://www.activestate.com, 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. The Perl tool Data::Dumper used with the syntax:

```
use Data::Dumper;
printer Dumper($seq);
```

can also be helpful for obtaining debugging information on perl objects.

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/, and 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

- 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-Parser-2.30
- XML-Twig-2.02
- XML-Writer-0.4
- Soap-Lite-0.52
- XML-DOM-1.37
- 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 bioinformatics 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- ftp://ftp-igbmc.u-strasbg.fr/pub/ClustalW/

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

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

for EMBOSS applications - http://www.hgmp.mrc.ac.uk/Software/EMBOSS/download.html

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 similiar 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.

The CPAN module can also be used to install all of the modules listed above in a single step as a "bundle" of modules, Bundle::BioPerl, eg

```
$>perl -MCPAN -e shell
cpan>install Bundle::BioPerl
<installation details....>
cpan>install B/BI/BIRNEY/bioperl-1.0.tar.gz
<installation details....>
cpan>quit
```

The disadvantage of this approach is that if there's a problem installing any individual module it may be a bit more difficult to address.

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 in *Bio*)

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,Mac). However, bioperl has not been widely tested under these OS's.

Todd Richmond has written of his experiences with BioPerl on MacOS 9 at http://bioperl.org/Core/mac-bioperl.html. There is also a description of bioperl on windows by Jurgen Pletinckx at http://www.bioperl.org/Core/windows-bioperl.html. (Note that currently these documents describe release 0.7.x of bioperl.) Minimal bioperl does run without problems on Mac OS X since it is a Unix system. However, external precompiled programs (eg NCBI local Blast) and other useful auxiliary programs such as perl-TK and ptkdb are in many cases not yet available under OS X.

Steve Cannon has compiled installation notes and suggestions for Bioperl on OS X online at http://www.tc.umn.edu/~cann0010/Bioperl_OSX_install.html.

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.

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) seven different “sequence objects” - Seq, PrimarySeq, LocatableSeq, LiveSeq, LargeSeq, SeqI, and SeqWithQuality. 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.1 Sequence objects: (Seq, RichSeq, SeqWithQuality, 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*, or in *Bio*.

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*.

RichSeq objects store additional annotations beyond those used by standard Seq objects. If you are using sources with very rich sequence annotation, you may want to consider using these objects which are described in section *III.7.1*. SeqWithQuality objects are used to manipulate sequences with quality data, like those produced by phred. These objects are described in section *III.7.4*, *Bio*, and in *Bio*.

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. See *Bio* for more details.

What is (for historical reasons) called a LocatableSeq object might be more appropriately called an "AlignedSeq" object. It is a Seq object which is part of a multiple sequence alignment. It has "start" and "end" positions indicating from where in a larger sequence it may have been extracted. It also may have "gap" symbols corresponding to the alignment to which it belongs. It is used by the alignment object SimpleAlign and other modules that use SimpleAlign objects (eg AlignIO.pm, pSW.pm). 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 you input an alignment data file using AlignIO. However if you need to input a sequence alignment by hand (eg to build a SimpleAlign object), you will need to input the sequences as LocatableSeqs. Other sources of information include *Bio*, *Bio*, *Bio*, and *Bio*.

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

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, LiveSeq does implement the SeqI interface (see below). Consequently, most methods available for Seq objects will work fine with LiveSeq objects. Section *III.7.2* and *Bio* contain further discussion of LiveSeq objects.

SeqI objects are Seq "interface objects" (see section *II.4* and *Bio*). 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.2 Alignment objects (SimpleAlign)

Early versions of bioperl used both UnivAln and SimpleAlign objects to represent and manipulate alignments but as of v. 1.0 only SimpleAlign.pm is supported. This module allows the user to convert between alignment formats as well as more sophisticated operations, like extracting specific regions of the alignment and generating consensus sequences. For more information see section *III.5.4* and *Bio*.

II.3 Location objects

A Location object is designed to be associated with a Sequence Feature object to indicate where on a larger structure (eg a chromosome or contig) the feature can be found. The reason why this simple concept has evolved in a collection of rather complicated objects is that

1) Some objects have multiple locations or sub-locations (eg a gene's exons may have multiple start and stop locations) 2) In unfinished genomes, the precise locations of features is not known with certainty.

Bioperl's various Location objects address these complications. In addition there are "CoordinatePolicy" objects that allow the user to specify how to measure the "length" of a feature if its precise start and end coordinates are not know. In most cases, you will not need to worry about these complications if you are using bioperl to handle simple features with well-defined start and stop locations. However, if you are using bioperl to annotate partially or unfinished genomes or to read annotations of such genomes with bioperl, understanding the various Location objects will be important. See the documentation of the various modules in the Bio::Locations directory or *Bio* for more information.

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).

For more discussion of design and development issues please see the biodesign.pod file.

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',
                       '-alphabet' => '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();
  # this returns a Seq object :
  $seq1 = $gb->get_Seq_by_id('MUSIGHBA1');
  # this returns a Seq object :
  $seq2 = $gb->get_Seq_by_acc('AF303112'))
  # this returns a SeqIO object :
  $seqio = $gb->get_Stream_by_batch([ qw(J00522 AF303112 2981014)]));
```

Bioperl currently supports sequence data retrieval from the genbank, genpept, RefSeq, swissprot, and EMBL databases. See *Bio*, *Bio*, *Bio*, *Bio* and *Bio* for more information. A user can also specify a different database mirror for a database - this is especially relevent for the SwissProt resource where there are many ExPaSy mirrors. There are also configuration options for specifying local proxy servers for those behind firewalls.

The retrieval of NCBI RefSeqs sequences is supported through a special module called Bio::DB::RefSeq which actually queries an EBI server. Please see *Bio* before using it as there are some caveats with RefSeq retrieval. RefSeq ids in Genbank begin with “NT_”, “NC_”, “NG_”, “NM_”, “NP_”, “XM_”, “XR_”, or “XP_” (for more information see http://www.ncbi.nlm.nih.gov/LocusLink/refseq.html). Bio::DB::GenBank can be used to retrieve entries corresponding to these ids but bear in mind that these are not Genbank entries, strictly speaking. See *Bio* for special details on retrieving entries beginning with “NT_”, these are specially formatted “CONTIG” entries.

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/.

An additional module is available for accessing remote databases, BioFetch, which queries the dbfetch script at EBI. The available databases are EMBL, GenBank, or SWALL, and the entries can be retrieved in different formats as objects or streams (SeqIO objects), or as “tempfiles”. See *Bio* for the details.

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

Alternately, bioperl permits indexing local sequence data files by means of the Bio::Index or Bio::DB::Fasta objects. The following sequence data formats are supported by Bio::Index: 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 in the scripts/ directory, bpindex.pl and bpfetch.pl. These scripts can be used as templates to develop customized local data-file indexing systems.

Bioperl also supplies Bio::DB::Fasta as a means to index and query Fasta format files. It's similar in spirit to Bio::Index::Fasta but offers more methods, eg

```
  use Bio::DB::Fasta;
  $db = Bio::DB::Fasta->new($file);  # one file or many files
  $seqstring = $db->seq($id);        # get a sequence as string
  $seqobj = $db->get_Seq_by_id($id); # get a PrimarySeq obj
  $desc = $db->header($id);          # get the header, or description, line
```

This module also offers the user the ability to designate a specific string within the fasta header as the desired id, such as the gi number within the string "gi|4556644|gb|X45555" (use the -makeid option for this capability). See *Bio* for more information on this fully-featured module.

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 a number of formats: Fasta, EMBL, GenBank, Swissprot, PIR, GCG, SCF, phd/phred, Ace, or raw (plain sequence). 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>;
```

If the “-format” argument isn’t used then Bioperl will guess the format based on the file’s suffix in a case-insensitive manner. Here are the current interpretations:

```
Format   Suffixes



fasta    fasta|fast|seq|fa|fsa|nt|aa
genbank  gb|gbank|genbank
scf      scf
pir      pir
embl     embl|ebl|emb|dat
raw      txt
gcg      gcg
ace      ace
bsml     bsm|bsml
swiss    swiss|sp
phd      phd|phred
```

For more information see *Bio*.

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, clustalw, msf/gcg, water (from EMBOSS, see *III.3.6*), needle (from EMBOSS, see *III.3.6*) 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. See *Bio* and section *III.5.4* for more information.

III.3 Manipulating sequences

Bioperl contains many modules with functions for sequence analysis. And if you cannot find the function you want in bioperl you may be able to find it in EMBOSS, which is accessible through bioperl (see *III.3.6*).

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->alphabet();   # one of 'dna','rna','protein'
$seqobj->primary_id(); # a unique id for this sequence irregardless
                       # of its display_id or accession number
$seqobj->desc()        # a description of the sequence
```

It is worth mentioning that some of these values correspond to specific fields of given formats. For example, the display_id method returns the LOCUS name of a Genbank entry, the (\S+) following the > character in a Fasta file, the ID from a SwissProt file, and so on. The `desc()` method will return the DEFINITION line of a Genbank file, the line following the display_id in a Fasta file, and the DE field in a SwissProt file.

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. A general description of the object can be found in *Bio*, and a description of related, top-level "annotation" is found in *Bio*.

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. See *Bio* for more information.

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 type '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');
```

See *Bio* for related details.

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, 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. See *Bio* and *Bio* for more information.

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 more than 150 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. See *Bio* for details.

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". Please see *Bio* for details.

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 Bio::Tools::OddCodes.

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 in *Bio* 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 seq_pattern.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.
```

More detail can be found in *Bio*.

III.3.6 Sequence manipulation using the Bioperl EMBOSS interface (Tools::Run::EMBOSSApplication)

EMBOSS (European Molecular Biology Open Source Software) is an extensive collection of sequence analysis programs written in the C programming language, from http://www.uk.embnet.org/Software/EMBOSS. There are a number of algorithms in EMBOSS that are not found in "Bioperl proper" (eg. calculating DNA melting temperature, finding repeats, identifying prospective antigenic sites) so if you if you cannot find the function you want in bioperl you might be able to find it in EMBOSS.

EMBOSS programs are usually called from the command line but bioperl provides a Perl "wrapper" for EMBOSS function calls so that they can be executed from within a Perl script. Of course, the EMBOSS package must be installed for the Bioperl wrapper to function.

In the future, it is planned that Bioperl EMBOSS objects will return appropriate Bioperl objects to the calling script in addition to generating standard EMBOSS reports. This functionality is being initially implemented with the EMBOSS sequence alignment programs, so that they will return SimpleAlign objects in a manner similar to the way the Bioperl modules TCoffee.pm and Clustalw.pm work (see section *III.5.4* for a discussion of SimpleAlign).

An example of the Bioperl EMBOSS wrapper where a file is returned would be:

```
$factory = new Bio::Factory::EMBOSS;
$compseqapp = $factory->program('compseq');
%input = ( -word     => 4,
           -sequence => $seqObj,
           -outfile  => $compseqoutfile );
$compseqapp->run(\%input);
$seqio = Bio::SeqIO->new( -file => $compseqoutfile ); # etc...
```

Note that a Seq object was used as input. The EMBOSS object can also accept a file name as input, eg

```
-sequence => "inputfasta.fa"
```

Some EMBOSS programs will return strings, others will create files that can be read directly using Bio::SeqIO (section *III.2.1*), as in the example above. It's worth mentioning that the AlignIO module can use files from EMBOSS's water and needle as input (see *III.2.2*) to create AlignIO objects.

III.3.7 Sequence manipulation without creating Bioperl "objects" (Perl.pm)

Using the Bio::Perl.pm module, it is possible to manipulate sequence data in Bioperl without explicitly creating Seq or SeqIO objects. This feature may ease the Bioperl learning curve for new users unfamiliar or uncomfortable with using Perl objects. However, only limited data manipulation are supported in this mode. In addition, each method (i.e. function) that will be used by the script must be explicitly declared in the initial “use directive”. For example a simple data format conversion and sequence manipulation could be performed as follows - note that no “new” methods are called and that no Seq or SeqIO objects are created:

```
use Bio::Perl qw( get_sequence );
# get a sequence from a database (assummes internet connection)
$seq_object = get_sequence('swissprot',"ROA1_HUMAN");
# $seq_object is Bio::Seq object, so the following methods work
$seq_id  = $seq_object->display_id;
$seq_as_string = $seq_object->seq();
```

For more details see *Bio*

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 Bio::Tools::Run::StandAloneBlast offers the ability to wrap local calls to blast from within perl. All of the currently available options of NCBI Blast (eg PSI-BLAST, 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::Run::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 Seq object or an array of 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 *Bio* documentation for details. In addition, the script standaloneblast.pl in the examples/ directory contains descriptions of various possible applications of the StandAloneBlast object. This script shows how the blast report object can access a blast parser directly, eg

```
while (my $sbjct = $blast_report->nextSbjct){
   while (my $hsp = $sbjct->nextHSP){
      print $hsp->score . " " . $hsp->subject->seqname . "\n";
   }
}
```

See the section *III.4.4* on parsing BLAST reports with Bio::Tools::BPlite, below, or *Bio* for details.

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

Bioperl supports remote execution of blasts at NCBI by means of the RemoteBlast object. (Note: remote execution of blasts is also possible using the bioperl Bio::Tools::Blast object. However this Blast object is no longer supported and its interface - especially for running Blasts is somewhat complicated. Consequently, the user is advised to use the Bio::Tools::Run::RemoteBlast object for this purpose - unless you really know what you're doing!)

A skeleton script to run a remote blast might look as follows:

```
$remote_blast = Bio::Tools::Run::RemoteBlast->new(
         '-prog' => 'blastp','-data' => 'ecoli','-expect' => '1e-10' );
$r = $remote_blast->submit_blast("t/data/ecolitst.fa");
while (@rids = $remote_blast->each_rid ) {
    foreach $rid ( @rids ) {$rc = $remote_blast->retrieve_blast($rid);}}
```

Note that the script has to be broken into two parts. The actual Blast submission and the subsequent retrieval of the results. At times when the NCBI Blast is being heavily used, the interval between when a Blast submission is made and when the results are available can be substantial.

The object `$rc` would contain the blast report that could then be parsed with Bio::Tools::BPlite or Bio::Tools::Blast. Note that to make this script actually useful, one should add details such as checking return codes from the Blast to see if it succeeded and and a "sleep" loop to wait between consecutive requests to the NCBI server. See example 21 in the demonstration script in the appendix to see some working code you could use, or *Bio* for details.

It should also be noted that the syntax for creating a remote blast factory is slightly different from that used in creating StandAloneBlast, Clustalw, and T-Coffee factories. Specifically RemoteBlast requires parameters to be passed with a leading hyphen, as in '-prog' => 'blastp', while the other programs do not pass parameters with a leading hyphen.

III.4.3 Parsing BLAST and FASTA reports with Search and SearchIO

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 several different objects - Search.pm / SearchIO.pm, BPlite.pm (along with its minor modifications, BPpsilite and BPbl2seq) and Blast.pm for parsing Blast reports. Search and SearchIO which are new in Bioperl 1.0 and are now the principal Bioperl interfaces for Blast (and FASTA) report parsing are described in this section. The older BPlite and Blast.pm objects are described in section *III.4.4*.

The Search and SearchIO modules provide a uniform interface for parsing sequence-similarity-search reports generated by BLAST (in standard and BLAST XML formats), PSIBLAST and FASTA. In the future, it is envisioned that the Search/SearchIO syntax will be extended to provide a uniform interface to a wider range of report parsers including parsers for HMMer and Genscan.

Parsing sequence-similarity reports with Search and SearchIO is straightforward. Initially a SearchIO object specifies a file containing the `report(s).` The method next_result reads the next report into a Search object in just the same way that the next_seq method of SeqIO reads in the next sequence in a file into a Seq object.

Once a report (i.e. a Search object) has been read in and is available to the script, the report's overall attributes (e.g. the query) can be determined and its individual "hits" can be accessed with the next_hit method. Individual high-scoring-pairs for each hit can then be accessed with the next_hsp method. Except for the additional syntax required to enable the reading of multiple reports in a single file, the remainder of the Search/SearchIO parsing syntax is very similar to that of the BPlite and Blast.pm objects it is intended to replace. Sample code to read a BLAST report might look like this:

```
# Get the report
$searchio = new Bio::SearchIO ('-format' => 'blast',
                               '-file'   => $blast_report);
$result = $searchio->next_result;
```

```
# Get info about the entire report
$result->database_name;
$algorithm_type =  $result->algorithm;
```

```
# get info about the first hit
$hit = $result->next_hit;
$hit_name = $hit->name ;
```

```
# get info about the first hsp of the first hit
$hsp = $hit->next_hsp;
$hsp_start = $hsp->query->start;
```

For more details on parsing with Search/SearchIO see the next section on BPlite and Blast.pm (which uses very similar syntax) as well as the Search and SearchIO documentation: *Bio*, *Bio*, *Bio*, *Bio*, and *Bio*.

There is also sample code is the searchio subdirectory of the Bio/examples directory which illustrates the use of the Search parser.

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

Bioperl's older BLAST report parsers - BPlite, BPpsilite, BPbl2seq and Blast.pm - are expected to be phased out over a period of time. Since a considerable amount of legacy Bioperl scripts has been written which heavily use these objects, they are likely to remain within Bioperl for some time.

Much of the user interface of BPlite (and to a lesser degree Blast.pm) is very similar to that of Search. However accessing the next hit or HSP uses methods called next_Sbjct and next_HSP, respectively - in contrast to Search's next_hit and next_hsp.

BPlite (with its relatives BPpsilite and BPbl2seq) 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 BPlite contains the functionality that you need, (unless you need to do HSP tiling or to implement an arbitrary filter function 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 Bio::Tools::BPlite(-fh=>\*STDIN);
```

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

A complete description of the module can be found in *Bio*.

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. See *Bio* and *Bio* for details.

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 Bio::Tools::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; }
}
```

See *Bio* for details.

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, one 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/data/dblseq.out",
                                    -queryname => "ALEU_HORVU");
$hsp = $report->next_feature;
$answer=$hsp->score;
```

In addition, since there will only be (at most) one "subject" (hit) in a bl2seq report one should use the method $report->next_feature, rather than $report->nextSbjct->nextHSP to obtain the next high scoring pair. See *Bio* for more details.

Blast.pm

The parser contained within the Bio::Tools::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/data/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.

A 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 );
```

Another useful feature of Blast.pm is "HSP tiling". With HSP tiling, if a Blast hit has more than one HSP, Blast.pm has the ability to merge overlapping HSPs into contiguous blocks. This enables one to sum data across all HSPs without counting data from overlapping regions multiple times. HSP tiling is performed automatically when invoking methods that rely on tiled data such as frac_identical and frac_conserved. For more information on HSP tiling see the documentation in *Bio* and *Bio*.

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 section *III.4.4*) or the Search/SearchIO parsers (section *III.4.3*) are recommended for most blast parsing within bioperl.

See *Bio* for more information.

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 Model (HMM) program that (among other capabilities) enables sequence similarity searching, from http://hmmer.wustl.edu. 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";
    }
}
```

Additional methods are described in *Bio*.

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 Bio::AlignIO (see section below, *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 script psw.pl in the examples/ directory and the documentation in *Bio*.

An alterative way to get Smith-Waterman alignments can come from the EMBOSS program 'water'. This can produce an output file that bioperl can read in with the AlignIO system

```
use Bio::AlignIO;
my $in = new Bio::AlignIO(-format => 'emboss', -file => 'filename');
my $aln = $in->next_aln();
```

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 Bio::AlignIO file format reader as follows:

```
$factory = Bio::Tools::Run::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 *Bio* and *Bio* 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 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 *Bio* and *Bio*. The user is also encouraged to examine the script clustalw.pl in the examples/ directory.

III.5.4 Manipulating / displaying alignments (SimpleAlign)

As described in section *II.2*, bioperl previously included two alignment objects, SimpleAlign and UnivAln, but UnivAln.pm is not supported as of v. 1.0. SimpleAlign objects are produced by bioperl alignment creation objects (eg Clustalw.pm, BLAST's bl2seq, and pSW) and they can read and write multiple alignment formats via AlignIO.

Some of the manipulations possible with SimpleALign include:

- `slice()`: Obtaining an alignment "slice", that is, a subalignment inclusive of specified start and end columns. Sequences with no residues in the slice are excluded from the new alignment and a warning is printed.
- `column_from_residue_number()`: Finding column in an alignment where a specified residue of a specified sequence is located.
- `consensus_string()`: Making a consensus string. This method includes an optional threshold parameter, so that positions in the alignment with lower percent-identity than the threshold are marked by "?"'s in the consensus
- `percentage_identity()`: A fast method for calculating the average percentage identity of the alignment
- `consensus_iupac()`: Making a consensus using IUPAC ambiguity codes from DNA and RNA.

Skeleton code for using some of these features is shown below. More detailed, working code is in Demo example 14 and in align_on_codons.pl in the scripts directory. Additional documentation on methods can be found in *Bio* and *Bio*.

```
use Bio::SimpleAlign;
```

```
$aln = Bio::SimpleAlign->new('t/data/testaln.dna');
$threshold_percent = 60;
$consensus_with_threshold = $aln->consensus_string($threshold_percent);
$iupac_consensus = $aln->consensus_iupac();   # dna/rna alignments only
$percent_ident = $aln->percentage_identity;
$seqname = '1433_LYCES';
$pos = $aln->column_from_residue_number($seqname, 14);
```

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

Automated searching for putative genes, coding sequences, sequence-tagged-sites (STS's) 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 feature searching programs currently exist. Each produces reports containing predictions that must be read manually or parsed by automated report readers.

Parsers for six widely used gene prediction programs - Genscan, Sim4, Genemark, Grail, 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; see *Bio*, *Bio*, *Bio*, *Bio*, *Bio*, and *Bio* for further details.

```
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();
```

See *Bio* and *Bio* for more details.

```
use Bio::Tools::Sim4::Results;
$sim4 = new Bio::Tools::Sim4::Results(-file => 't/data/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();
```

See *Bio* and *Bio* for more information.

A parser for the ePCR program is also available. The ePCR program identifies potential PCR-based sequence tagged sites (STSs) For more details see the documentation in *Bio*. A sample skeleton script for parsing an ePCR report and using the data to annotate a genomic sequence might look like this:

```
use Bio::Tools::EPCR;
use Bio::SeqIO;
$parser = new Bio::Tools::EPCR(-file => 'seq1.epcr');
$seqio = new Bio::SeqIO(-format => 'fasta', -file => 'seq1.fa');
$seq = $seqio->next_seq;
while( $feat = $parser->next_feature ) {
      # add EPCR annotation to a sequence
      $seq->add_SeqFeature($feat);}
```

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 fea-
tures
```

See *Bio* and *Bio* as starting points for further exploration, and see the scripts/gff2ps.pl script.

In general, storing and retrieving feature information should be straightforward. However, one potential trap relates to features whose location is either "split" - as in a multi-exon gene - or "fuzzy" - as when genomic coordinates are not yet known with certainty. In these cases, the SeqFeature objects need to be built with one of the alternate Location objects described in Section *II.3*.

If more detailed annotation than available in Seq objects is required, the RichSeq object may be used. It is applicable in particular to database sequences (EMBL, GenBank and Swissprot) with detailed annotations. Sample usage might be:

```
@secondary   = $richseq->get_secondary_accessions;
$division    = $richseq->division;
@dates       = $richseq->get_dates;
$seq_version = $richseq->seq_version;
```

See *Bio* for more details.

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

This interface extends the Bio::SeqI interface to give additional functionality to sequences with richer data sources, in particular from database sequences (EMBL, GenBank and Swissprot).

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* or *Bio* 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/data/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 section *II.1* or *Bio*). Consequently syntax for using LiveSeq objects is familiar although a modified version of SeqIO called Bio::LiveSeq::IO::Bioperl needs to be used to actually load the data, eg:

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

See *Bio* for more details.

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)

The Mutation object allows for a basic description of a sequence change in the DNA sequence of a gene. The Mutator object 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 level.

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 *Bio* and *Bio* as well as the original documentation for the "Computational Mutation Expression Toolkit" project at http://www.ebi.ac.uk/mutations/toolkit/.

III.7.4 Incorpotating quality data in sequence annotation (SeqWithQuality)

SeqWithQuality objects are used to describe sequences with very specific annotations - that is, data quality annotaions. Data quality information is important for documenting the reliability of base "calls" in newly sequenced or otherwise questionable sequence data. The quality data is contained within a Bio::Seq::PrimaryQual object. Syntax for using SeqWithQuality objects is as follows:

```
  # first, make a PrimarySeq object
  $seqobj = Bio::PrimarySeq->new
          ( -seq => 'atcgatcg',            -id  => 'GeneFragment-12',
            -accession_number => 'X78121', -alphabet => 'dna');
  # now make a PrimaryQual object
  $qualobj = Bio::Seq::PrimaryQual->new
         ( -qual => '10 20 30 40 50 50 20 10', -id  => 'GeneFragment-
12',
           -accession_number => 'X78121',      -alphabet => 'dna');
  # now make the SeqWithQuality object
  $swqobj = Bio::Seq::SeqQithQuality->new
          ( -seq  => $seqobj, -qual => $qualobj);
  # Now we access the sequence with quality object
  $swqobj->id(); # the id of the SeqWithQuality object may not match the
                 # id of the sequence or of the quality
  $swqobj->seq(); # the sequence of the SeqWithQuality object
  $swqobj->qual(); # the quality of the SeqWithQuality object
```

A SeqWithQuality object is created automatically when phred output, a `*phd` file, is read by Seqio, eg

```
  $seqio = Bio::SeqIO->new(-file=>"my.phd",-format=>"phd");
  # or just 'Bio::SeqIO->new(-file=>"my.phd")'
  $seqWithQualObj = $seqio->next_seq;
```

See *Bio* for a detailed description of the methods, *Bio*, and *Bio*.

III.7.5 Sequence XML representations - generation and parsing (SeqIO::game, SeqIO::bsml)

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 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/data/test.game',
                       '-format' => 'game');
$seq = $str->next_primary_seq();
$id = $seq->id;
@feats = $seq->all_SeqFeatures();
$first_primary_tag = $feats[0]->primary_tag;
```

Additional XML formats to describe sequences and their annotations have been created. BSML and AGAVE are two additional formats that have been created in the last year. Bioperl currently only supports BSML through the SeqIO system at this time. Usage is similar to other SeqIO parsing.

```
$str = Bio::SeqIO->new('-file'=> 'bsmlfile.xml',
                       '-format' => 'bsml');
$seq = $str->next_primary_seq();
$id = $seq->id;
@feats = $seq->all_SeqFeatures();
$first_primary_tag = $feats[0]->primary_tag;
```

III.8 Representing non-sequence data in Bioperl: structures, trees and maps

Though bioperl has its roots in describing and searching nucleotide and protein sequences it has also branched out into related fields of study, such as protein structure, phylogenetic trees and genetic maps.

III.8.1 Using 3D structure objects and reading PDB files (StructureI, Structure::IO)

A StructureIO object can be created from one or more 3D structures represented in Protein Data Bank, or pdb, format (see http://www.rcsb.org/pdb for details).

StructureIO objects allow access to a variety of related Bio:Structure objects. An Entry object consist of one or more Model objects, which in turn consist of one or more Chain objects. A Chain is composed of Residue objects, which in turn consist of Atom objects. There's a wealth of methods, here are just a few:

```
$structio = Bio::Structure::IO->new( -file => "1XYZ.pdb");
$struc = $structio->next_structure; # returns an Entry object
$ann = $struc->annotation; # returns a Bio::Annotation object
$pseq = $struc->seqres;    # returns a PrimarySeq object, thus
$pseq->subseq(1,20);              # returns a sequence string
@atoms = $struc->get_atoms($res); # Atom objects, given a Residue
@xyz = $atom->xyz;                # the 3D coordinates of the atom
```

These lines show how one has access to a number of related objects and methods. For examples of typical usage of these modules, see the scripts in the examples/structure subdirectory. Also see *Bio*, *Bio*, *Bio*, *Bio*, *Bio*, and *Bio* for more information.

III.8.2 Tree objects and phylogenetic trees (Tree::Tree, TreeIO)

Bioperl Tree objects can store data for all kinds of computer trees and are intended especially for phylogenetic trees. Nodes and branches of trees can be individually manipulated. The TreeIO object is used for stream I/O of tree objects. Currently only phylip/newick tree format is supported. Sample code might be:

```
$treeio = new Bio::TreeIO( -format => 'newick', -file   => $treefile);
$tree = $treeio->next_tree;   # get the tree
@nodes = $tree->get_nodes;    # get all the nodes
$tree->get_root_node()->each_Descendent();  # get descendents of root node
```

See *Bio* and *Bio* for details.

III.8.3 Map objects for manipulating genetic maps (Map::MapI, MapIO)

Bioperl map objects can be used to describe any type of biological map data including genetic maps, STS maps etc. Map I/O is performed with the MapIO object which works in a similar manner to the SeqIO, SearchIO and similar I/O objects described previously. In principle, Map I/O with various map data formats can be performed. However currently only "mapmaker" format is supported. Manipulation of genetic map data with Bioperl Map objects might look like this:

```
$mapio = new Bio::MapIO( '-format' => 'mapmaker', '-file' => $mapfile);
$map = $mapio->next_map;  # get a map
$maptype =  $map->type ;
foreach  $marker ( $map->each_element ) {
  $marker_name =  $marker->name ;  # get the name of each map marker
}
```

See *Bio* and *Bio* for more information.

III.8.4 Bibliographic objects for querying bibliographic databases (Biblio)

Bio::Biblio objects are used to query bibliographic databases, such as MEDLINE. The associated modules are built to work with OpenBQS-compatible databases (see http://industry.ebi.ac.uk/openBQS). A Bio::Biblio object can execute a query like:

```
my $collection = $biblio->find ('brazma', 'authors');
while ( $collection->has_next ) {
    print $collection->get_next;
}
```

See *Bio* or the examples/biblio.pl script for details.

III.8.5 Graphics objects for representing sequence objects as images (Graphics)

A user may want to represent Seq objects and their SeqFeatures graphically. The Bio::Graphics::* modules use Perl's GD.pm module to create a PNG or GIF image given the SeqFeatures (Section *III.7.1*) contained within a Seq object.

These modules contain numerous methods to dictate the sizes, colors, labels, and line formats within the image. See *Bio*, *Bio*, or the scripts/render_sequence.pl script for more information.

The Genquire application also provides ways to graphically represent Seq objects (see Section *IV.6*).

III.9 Bioperl alphabets

Bioperl modules use the standard extended single-letter genetic alphabets to represent nucleotide and amino acid sequences.

In addition to the standard alphabet, the following symbols are also acceptable in a biosequence:

```
?  (a missing nucleotide or amino acid)
-  (gap in sequence)
```

III.9.1 Extended DNA / RNA alphabet

```
(includes symbols for nucleotide ambiguity)
------------------------------------------
Symbol       Meaning      Nucleic Acid
------------------------------------------
 A            A           Adenine
 C            C           Cytosine
 G            G           Guanine
 T            T           Thymine
 U            U           Uracil
 M          A or C
 R          A or G
 W          A or T
 S          C or G
 Y          C or T
 K          G or T
 V        A or C or G
 H        A or C or T
 D        A or G or T
 B        C or G or T
 X      G or A or T or C
 N      G or A or T or C


IUPAC-IUB SYMBOLS FOR NUCLEOTIDE NOMENCLATURE:
  Cornish-Bowden (1985) Nucl. Acids Res. 13: 3021-3030.
```

III.9.2 Amino Acid alphabet

```
------------------------------------------
Symbol   Meaning
------------------------------------------
A        Alanine
B        Aspartic Acid, Asparagine
C        Cystine
D        Aspartic Acid
E        Glutamic Acid
F        Phenylalanine
G        Glycine
H        Histidine
I        Isoleucine
K        Lysine
L        Leucine
M        Methionine
N        Asparagine
P        Proline
Q        Glutamine
R        Arginine
```

```
S       Serine
T       Threonine
V       Valine
W       Tryptophan
X       Unknown
Y       Tyrosine
Z       Glutamic Acid, Glutamine
*       Terminator


    IUPAC-IUP AMINO ACID SYMBOLS:
    Biochem J. 1984 Apr 15; 219(2): 345-373
    Eur J Biochem. 1993 Apr 1; 213(1): 2
```

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

There are several "sister projects" to bioperl currently under development. These include biocorba, biopython, biojava, EMBOSS, Ensembl, and Genquire / 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, see the biocorba project website at http://biocorba.org/. The Bioperl BioCORBA server and client bindings are available in the bioperl-corba-server and bioperl-corba-client bioperl CVS repositories respecitively. (see http://cvs.bioperl.org for more information).

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 EMBOSS

EMBOSS is another open source project with similar goals to bioperl. However EMBOSS code is implemented in C and has been designed for standalone execution on the Unix command line, rather than for incorporation into a user script or program. EMBOSS includes a wide array of useful bioinformatics functions similar to those of the GCG package after which it was designed. A bioperl interface to the EMBOSS functions has been partially completed. When this interface is complete, it will be possible to access EMBOSS functions as though they were bioperl objects (in a manner similar to how the StandAloneBlast bioperl module enables access to performing Blast searches within bioperl). For more information on EMBOSS, refer to http://www.hgmp.mrc.ac.uk/Software/EMBOSS/. The principal bioperl interface object to EMBOSS is described in *Bio*.

IV.4 Ensembl and bioperl-db

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. 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/.

Bioperl-db is a relatively new project intended to transfer some of Ensembl's capability of integrating bioperl syntax with a standalone Mysql database (http://www.mysql.com) to the bioperl code-base. More details on bioperl-db can be found in the bioperl-db CVS directory at http://cvs.bioperl.org/cgi-bin/viewcvs/viewcvs.cgi/bioperl-db/?cvsroot=bioperl. It is worth mentioning that most of the bioperl objects mentioned above map directly to tables in the bioperl-db schema. Therefore object data such as sequences, their features, and annotations can be easily loaded into the databases, as in

```
$loader->store($newid,$seqobj)
```

Similarly one can query the database in a variety of ways and retrieve arrays of Seq objects. See biodatabases.pod, *Bio*, *Bio*, and *Bio* for examples.

IV.5 GFF format and Bio::DB::GFF*

The Bio::DB::GFF module provides access to relational databases constructed from data files in GFF format. This file type is well suited to sequence annotation because it allows the ability to describe entries in terms of parent-child relationships (see http://www.sanger.ac.uk/software/GFF for details). Like bioperl-db, above, the current implementation uses mysql (http://www.mysql.com).

The module accesses not only by id but by annotation type and position or range. Those who would like to explore bioperl as a means to overlay nucleotide sequence, protein sequence, features, and annotations should take a close look at *Bio*and the load_gff.pl, bulk_load_gff.pl, gadfly_to_gff.pl, and sgd_to_gff.pl scripts in the scripts/Bio-DB-GFF directory.

IV.6 Genquire, the Annotation Workbench and bioperl-gui

The Annotation Workbench and Genquire were developed at the Plant Biotechnology Institute of the National Research Council of Canada. This is an integrated graphical suite of tools in Perl for examining a sequence, predicting gene structure, and creating annotations. Information about Genquire is available at http://bioinformatics.org/project/?group_id=99. With Genquire and bioperl-gui one can display a Bioperl Seq object graphically. You can download the current version of the gui software from the bioperl-gui CVS directory at http://cvs.bioperl.org/cgi-bin/viewcvs/viewcvs.cgi/bioperl-gui/?cvsroot=bioperl. You will also need Tcl/Tk.

V.1 Appendix: Finding out which methods are used by which Bioperl Objects

At numerous places in the tutorial, the reader is directed to the “documentation included with each of the modules.” As was mentioned in the introduction, it is sometimes not easy in perl to determine the appropriate documentation to look for, because objects inherit methods from other objects (and the relevant documentation will be stored in the object from which the method was inherited.)

For example, say you wanted to find documentation on the “parse” method of the object Genscan.pm. You would not find this documentation in the code for Genscan.pm, but rather in the code for AnalysisResult.pm from which Genscan.pm inherits the parse method!

So how would you know to look in AnalysisResult.pm for this documentation? The easy way is to use the special function “option 100” in the bptutorial script. Specifically if you run:

```
> perl -w bptutorial.pl 100 Bio::Tools::Genscan
```

you will receive the following output:

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


Methods taken from package Bio::Root::IO
catfile   close   gensym   new   qualify   qualify_to_ref
rmtree   tempdir   tempfile   ungensym


Methods taken from package Bio::Root::RootI
DESTROY   stack_trace   stack_trace_dump   throw   verbose   warn


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


Methods taken from package Bio::Tools::AnalysisResult
analysis_date   analysis_method   analysis_method_version   analysis_query   anal-
ysis_subject   parse


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

From this output, it is clear exactly from which object each method of Genscan.pm is taken, and, in particular that "parse" is taken from the package Bio::Tools::AnalysisResult.

With this approach you can easily determine the source of any method in any bioperl object.

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.