HOWTO:Beginners

From BioPerl

Contents

Authors

Brian Osborne

briano at bioteam.net (mailto:briano@bioteam.net)

Copyright

Abstract

This is a HOWTO that talks about using Bioperl, for biologists who would like to learn more about writing their own bioinformatics scripts using Bioperl. What is Bioperl? It is an open source bioinformatics toolkit used by researchers all over the world. If you're looking for a script built to fit your exact needs you probably won't find it in Bioperl. What you will find is a diverse set of Perl modules that will enable you to write your own script, and a community of people who are willing to help you.

Introduction

If you're a molecular biologist it's likely that you're interested in gene and protein sequences, and you study them in some way on a regular basis. Perhaps you'd like to try your hand at automating some of these tasks, or you're just curious about learning more about the programming side of bioinformatics. In this HOWTO you'll see discussions of some of the common uses of Bioperl, like sequence analysis with BLAST and retrieving sequences from public databases. You'll also see how to write Bioperl scripts that chain these tasks together, that's how you'll be able to do really powerful things with Bioperl.

You will also see some discussions of software concepts; this can't be avoided. The more you understand about programming the better but all efforts will be made to not introduce too much unfamiliar material. However, there will be an introduction to modularity, or objects. This is one of the aspects of the Bioperl package that you'll have to come to grips with as you attempt more complex tasks with your scripts.

One of the challenging aspects of learning a new skill is learning the jargon, and programming certainly has its share of interesting terms and buzz phrases. Be patient - remember that the programmers learning biology have had just as tough a task (if not worse - just ask them!).

Note: This HOWTO does not discuss a very nice module that's designed for beginners, Bio::Perl (http://search.cpan.org/search?query=Bio::Perl&mode=all) . The reason is that though this is an excellent introductory tool, it is not object-oriented and can't be extended. What we're attempting here is to introduce the core of Bioperl and show you ways to expand your new-found skills.

Installing Bioperl

Start at Installing Bioperl. Many of the letters to the bioperl-l mailing list concern problems with installation, and there is a set of concerns that come up repeatedly:

- On Windows, messages like: "`Error: Failed to download URL http://bioperl.org/DIST/GD.ppd`", or "`<some module> Not found`". The explanation is that Bioperl does not supply every accessory module that's necessary to run all of Bioperl. You'll need to search other repositories to install all of these accessory modules. See the Installing_Bioperl_on_Windows file for more information.
- On Unix, messages like "`Can't locate <some module>.pm in @INC...`". This means that Perl could not find a particular module and the explanation usually is that this module is not installed. See the Installing_Bioperl_for_Unix file for details.
- Seeing messages like "`Tests Failed`". If you see an error during installation consider whether this problem is going to affect your use of Bioperl. There are roughly 1000 modules in Bioperl, and ten times that many tests are run during the installation. If there's a complaint about GD it's only relevant if you want to use the Bio::Graphics (http://search.cpan.org/search?query=Bio::Graphics&mode=all) modules, if you see an error about some XML parser it's only going to affect you if you're reading XML files. Yes, you could try and make each and every test pass, but that may be a lot of work, with much of it fixing modules that aren't in BioPerl itself.

Getting Assistance

People will run into problems installing Bioperl or writing scripts using Bioperl, nothing unusual about that. If you need assistance the way to get it is to mail bioperl-l@bioperl.org. There are a good number of helpful people who regularly read this list but if you want their advice it's best to give sufficient detail.

Please include:

- The version of Bioperl you're working with.
- The platform or operating system you're using.
- What you are trying to do.
- The code that gives the error, if you're writing a script.
- Any error messages you saw.

Every once in a while a message will appear in bioperl-l coming from someone in distress that goes unanswered. The explanation is usually that the person neglected to include 1 or more of the details above, usually the script or the error messages.

Perl Itself

Here are a few things you might want to look at if you want to learn more about Perl:

- Learning Perl (http://www.oreilly.com/catalog/lperl2/) is the most frequently cited beginner's book.
- Perl in a Nutshell (http://www.oreilly.com/catalog/perlnut2/) is also good. Not much in the way of examples, but covers many topics succinctly.
- Perl's own documentation. Do "perldoc perl" from the command-line for an introduction. Perldoc can give you documentation of any module that is installed on your system: do "perldoc <modulename>" to view documentation of <modulename>. Try for instance (assuming Bioperl has been installed):

```
>perldoc Bio::SeqIO
```

Writing a script

Sometimes the trickiest part is this step, writing something and getting it to run, so this section attempts to address some of the more common tribulations.

In Unix when you're ready to work you're usually in the command-line or "shell" environment. First find out Perl's version by typing this command:

```
>perl -v
```

You will see something like:

```
This is perl, v5.10.0 built for cygwin-thread-multi-64int

Copyright 1987-2007, Larry Wall

Perl may be copied only under the terms of either the Artistic License or the
GNU General Public License, which may be found in the Perl 5 source kit.

Complete documentation for Perl, including FAQ lists, should be found on
this system using "man perl" or "perldoc perl".  If you have access to the
Internet, point your browser at http://www.perl.org/, the Perl Home Page.
```

Hopefully you're using Perl version 5.4 or higher, earlier versions may be troublesome. Now let's find out where the Perl program is located:

```
>which perl
```

This will give you something like:

```
>/bin/perl
```

Now that we know where Perl is located we're ready to write a script, and line 1 of the script will specify this location. You might be using some Unix word processor, emacs or vi, for example (nano or pico are other possible choices, very easy to use, but not found on all Unix machines unfortunately). If you're on Windows then Wordpad will work.

Start to write your script by entering something like:

```
>emacs seqio.pl
```

And make this the first line of the script:

```
#!/bin/perl
```

Creating a sequence, and an Object

Our first script will create a sequence. Well, not just a sequence, you will be creating a *sequence object*, since Bioperl is written in an object-oriented way. Why be object-oriented? Why introduce these odd or intrusive notions into software that should be *biological* or *intuitive*? The reason is that thinking in terms of modules or objects turns out to be the most flexible, and ultimately the simplest, way to deal with data as complex as biological data. Once you get over your initial skepticism, and have written a few scripts, you will find this idea of an object becoming a bit more natural.

One way to think about an object in software is that it is a container for data. The typical sequence entry contains different sorts of data (a sequence, one or more identifiers, and so on) so it will serve as a nice example of what an object can be.

All objects in Bioperl are created by specific Bioperl modules, so if you want to create an object you're also going to have to tell Perl which module to use. Let's add another line:

```
#!/bin/perl -w

use Bio::Seq;
```

This line tells Perl to use a module on your machine called "Bio/Seq.pm". We will use this Bio::Seq (http://search.cpan.org/search?query=Bio::Seq&mode=all) module to create a Bio::Seq (http://search.cpan.org/search?query=Bio::Seq&mode=all) object. The Bio::Seq (http://search.cpan.org/search?query=Bio::Seq&mode=all) module is one of the central modules in Bioperl. The analogous Bio::Seq (http://search.cpan.org/search?query=Bio::Seq&mode=all) object, or "Sequence object", or "Seq object", is ubiquitous in Bioperl, it contains a single sequence and associated names, identifiers, and properties. Let's create a very simple sequence object at first, like so:

```
#!/bin/perl -w

use Bio::Seq;

$seq_obj = Bio::Seq->new(-seq => "aaaatgggggggggggccccgtt",
                         -alphabet => 'dna' );
```

That's it! The variable `$seq_obj` is the Sequence object, a simple one, containing just a sequence. Note that the code tells Bioperl that the sequence is DNA (the choices here are 'dna', 'rna', and 'protein'), this is the wise thing to do. If you don't tell Bioperl it will attempt to guess the alphabet. Normally it guesses correctly but if your sequence has lots of odd or ambiguous characters, such as N or X, Bioperl's guess may be incorrect and this may lead to some problems.

Bio::Seq (http://search.cpan.org/search?query=Bio::Seq&mode=all) objects can be created manually, as above, but they're also created automatically in many operations in Bioperl, for example when alignment files or database entries or BLAST reports are parsed.

Any time you explicitly create an object, you will use this `new()` method. The syntax of this line is one you'll see again and again in Bioperl: the name of the object or variable, the module name, the `->` symbol, the method name new, some argument name like **-seq**, the **=>** symbol, and then the argument or value itself, like **aaaatgggggggggggccccgtt**.

Note: If you've programmed before you've come across the term "function" or "sub-routine". In object-oriented programming the term "method" is used instead.

The object was described as a data container, but it is more than that. It can also do work, meaning it can use or call specific methods taken from the module or modules that were used to create it. For example, the Bio::Seq module can access a method named `seq()` that will print out the sequence of Bio::Seq (http://search.cpan.org/search?query=Bio::Seq&mode=all) objects. You could use it like this:

```
#!/bin/perl -w

use Bio::Seq;

$seq_obj = Bio::Seq->new(-seq => "aaaatgggggggggggccccgtt", -alphabet => 'dna' );

print $seq_obj->seq;
```

As you'd expect, this script will print out **aaaatgggggggggggccccgtt**. That `->` symbol is used when an object calls or accesses its methods.

Let's make our example a bit more true-to-life, since a typical sequence object needs an identifier, perhaps a description, in addition to its sequence.

```
#!/bin/perl -w

use Bio::Seq;

$seq_obj = Bio::Seq->new(-seq => "aaaatgggggggggggccccgtt",
                         -display_id => "#12345",
                         -desc => "example 1",
                         -alphabet => "dna" );

print $seq_obj->seq();
```

aaaatgggggggggggccccgtt, **#12345**, and **example 1** are called "arguments" in programming jargon. You could say that this example shows how to pass arguments to the `new()` method.

Writing a sequence to a file

This next example will show how two objects can work together to create a sequence file. We already have a Sequence object, `$seq_obj`, and we will create an additional object whose responsibility it is to read from and write to files. This object is the SeqIO object, where IO stands for Input-Output. By using Bio::SeqIO (http://search.cpan.org/search?query=Bio::SeqIO&mode=all) in this manner you will be able to get input and make output for all of the sequence file formats supported by Bioperl (the SeqIO HOWTO has a complete list of supported formats). The way you create Bio::SeqIO (http://search.cpan.org/search?query=Bio::SeqIO&mode=all) objects is very similar to the way we used `new()` to create a Bio::Seq (http://search.cpan.org/search?query=Bio::Seq&mode=all) , or sequence, object:

```
use Bio::SeqIO;

$seqio_obj = Bio::SeqIO->new(-file => '>sequence.fasta', -format => 'fasta' );
```

Note that > in the `-file` argument. This character indicates that we're going to write to the file named "sequence.fasta", the same character we'd use if we were using Perl's `open()` function to write to a file. The `-format` argument, "fasta", tells the Bio::SeqIO (http://search.cpan.org/search?query=Bio::SeqIO&mode=all) object that it should create the file in fasta format.

Let's put our 2 examples together:

```
#!/bin/perl -w

use Bio::Seq;
use Bio::SeqIO;

$seq_obj = Bio::Seq->new(-seq => "aaaatgggggggggggccccgtt",
                                        -display_id => "#12345",
                                        -desc => "example 1",
                                        -alphabet => "dna" );

$seqio_obj = Bio::SeqIO->new(-file => '>sequence.fasta', -format => 'fasta' );

$seqio_obj->write_seq($seq_obj);
```

Let's consider that last `write_seq` line where you see two objects since this is where some neophytes start to get a bit nervous. What's going on there? In that line we handed or passed the Sequence object to the Bio::SeqIO (http://search.cpan.org/search?query=Bio::SeqIO&mode=all) object as an argument to its `write_seq` method. Another way to think about this is that we hand the Sequence object to the Bio::SeqIO (http://search.cpan.org/search?query=Bio::SeqIO&mode=all) object since Bio::SeqIO (http://search.cpan.org/search?query=Bio::SeqIO&mode=all) understands how to take information from the Sequence object and write to a file using that information, in this case in fasta format. If you run this script like this:

```
>perl seqio.pl
```

You should create a file called "sequence.fasta" that looks like this:

```
>#12345 example 1
aaaatgggggggggggccccgtt
```

Let's demonstrate the intelligence of the SeqIO - the example below shows what file content is created when the argument to "-format" is set to "genbank" instead of "fasta":

```
LOCUS       #12345                  23 bp    dna     linear   UNK
DEFINITION  example 1
ACCESSION   unknown
FEATURES             Location/Qualifiers
BASE COUNT        4 a      4 c     12 g      3 t
ORIGIN       1 aaaatggggg gggggggcccc gtt
//
```

Retrieving a sequence from a file

One beginner's mistake is to not use Bio::SeqIO (http://search.cpan.org/search?query=Bio::SeqIO&mode=all) when working with sequence files. This is understandable in some respects. You may have read about Perl's `open` function, and Bioperl's way of retrieving sequences may look odd and overly complicated, at first. But don't use `open`! Using `open()`, immediately forces you to do the parsing of the sequence file and this can get complicated very quickly. Trust the Bio::SeqIO (http://search.cpan.org/search?query=Bio::SeqIO&mode=all) object, it's built to open and parse all the common sequence formats, it can read and write to files, and it's built to operate with all the other Bioperl modules that you will want to use.

Let's read the file we created previously, "sequence.fasta", using Bio::SeqIO (http://search.cpan.org/search?query=Bio::SeqIO&mode=all) . The syntax will look familiar:

```
#!/bin/perl -w

use Bio::SeqIO;

$seqio_obj = Bio::SeqIO->new(-file => "sequence.fasta", -format => "fasta" );
```

One difference is immediately apparent: there is no **>** character. Just as with with the `open()` function this means we'll be reading from the "sequence.fasta" file. Let's add the key line, where we actually retrieve the Sequence object from the file using the `next_seq` method:

```
#!/bin/perl -w

use Bio::SeqIO;

$seqio_obj = Bio::SeqIO->new(-file => "sequence.fasta", -format => "fasta" );

$seq_obj = $seqio_obj->next_seq;
```

Here we've used the `next_seq()` method of the Bio::SeqIO (http://search.cpan.org/search?query=Bio::SeqIO&mode=all) object. When you use, or call, `next_seq()` the Bio::SeqIO (http://search.cpan.org/search?query=Bio::SeqIO&mode=all) object will get the next available sequence, in this case the first sequence in the file that was just opened. The Sequence object that's created, `$seq_obj`, is functionally just like the Sequence object we created manually in our first example. This is another idiom that's used frequently in Bioperl, the *next_<something>* method. You'll come across the same idea in the `next_aln` method of Bio::AlignIO (http://search.cpan.org/search?query=Bio::AlignIO&mode=all) (reading and writing alignment files) and the `next_hit` method of Bio::SearchIO (http://search.cpan.org/search?query=Bio::SearchIO&mode=all) (reading the output of sequence comparison programs such as BLAST and HMMER).

If there were multiple sequences in the input file you could just continue to call `next_seq()` in some loop, and SeqIO would retrieve the Seq objects, one by one, until none were left:

```
while ($seq_obj = $seqio_obj->next_seq){
    # print the sequence
```

```
    print $seq_obj->seq,"\n";
}
```

Do you have to supply a `-format` argument when you are reading from a file, as we did? Not necessarily, but it's the safe thing to do. If you don't give a format then the SeqIO object will try to determine the format from the file suffix or extension (and a list of the file extensions is in the SeqIO HOWTO). In fact, the suffix "fasta" is one that SeqIO understands, so `-format` is unnecessary above. Without a known suffix SeqIO will attempt to guess the format based on the file's contents but there's no guarantee that it can guess correctly for every single format.

It may be useful to tell SeqIO the alphabet of the input, using the `-alphabet` argument. What this does is to tell SeqIO not to try to determine the alphabet ("dna", "rna", "protein"). This helps because Bioperl may guess incorrectly (for example, Bioperl is going to guess that the protein sequence **MGGGGTCAATT** is DNA). There may also be odd characters present in the sequence that SeqIO objects to (e.g. "`-~?`"). Set `-alphabet` to a value when reading sequences and SeqIO will not attempt to guess the alphabet of those sequences or validate the sequences.

Retrieving a sequence from a database

One of the strengths of Bioperl is that it allows you to retrieve sequences from all sorts of sources, files, remote databases, local databases, regardless of their format. Let's use this capability to get a entry from Genbank (Bio::DB::GenBank (http://search.cpan.org/search?query=Bio::DB::GenBank&mode=all)). What will we retrieve? Again, a Sequence object. Let's choose our module:

```
use Bio::DB::GenBank;
```

We could also query SwissProt (Bio::DB::SwissProt (http://search.cpan.org /search?query=Bio::DB::SwissProt&mode=all)), GenPept (Bio::DB::GenPept (http://search.cpan.org /search?query=Bio::DB::GenPept&mode=all)), EMBL (Bio::DB::EMBL (http://search.cpan.org /search?query=Bio::DB::EMBL&mode=all)), SeqHound (Bio::DB::SeqHound (http://search.cpan.org /search?query=Bio::DB::SeqHound&mode=all)), Entrez Gene (Bio::DB::EntrezGene (http://search.cpan.org /search?query=Bio::DB::EntrezGene&mode=all)), or RefSeq (Bio::DB::RefSeq (http://search.cpan.org /search?query=Bio::DB::RefSeq&mode=all)) in an analogous fashion (e.g "use Bio::DB::SwissProt"). Now we'll create the object:

```
use Bio::DB::GenBank;

$db_obj = Bio::DB::GenBank->new;
```

In this case we've created a "database object" using the `new` method, but without any arguments. Let's ask the object to do something useful:

```
use Bio::DB::GenBank;

$db_obj = Bio::DB::GenBank->new;

$seq_obj = $db_obj->get_Seq_by_id(2);
```

The argument passed to the `get_Seq_by_id` method is an identifier, 2, a Genbank GI number. You could also use the `get_Seq_by_acc` method with an accession number (e.g. A12345) or `get_Seq_by_version` using a versioned accession number (e.g. A12345.2). Make sure to use the proper identifier for the method you use, the methods are not interchangeable.

Different Bioperl classes can query remote databases for single or multiple records. These should not be used in loops to retrieve large numbers of record; NCBI will block your IP if they feel you are abusing their services. There are much better/faster ways to do this: download a GenBank section and parse it directly, or use a BLAST-formatted database and fastacmd to get the seqs of interest in FASTA format.

Retrieving multiple sequences from a database

There are more sophisticated ways to query Genbank than this. This next example attempts to do something "biological", using the module Bio::DB::Query::GenBank (http://search.cpan.org/search?query=Bio::DB::Query::GenBank&mode=all) . Want all Arabidopsis topoisomerases from Genbank Nucleotide? This would be a reasonable first attempt:

```
use Bio::DB::Query::GenBank;

$query = "Arabidopsis[ORGN] AND topoisomerase[TITL] and 0:3000[SLEN]";
$query_obj = Bio::DB::Query::GenBank->new(-db    => 'nucleotide', -query => $query );
```

Note: This capability to query by string and field is only available for GenBank as of Bioperl version 1.5, queries to other databases, like Swissprot or EMBL, are limited to identifiers and accessions.

Here's another query example, this one will retrieve all *Trypanosoma brucei* ESTs:

```
$query_obj = Bio::DB::Query::GenBank->new(
                      -query   =>'gbdiv est[prop] AND Trypanosoma brucei [organism]',
                      -db      => 'nucleotide' );
```

You can find detailed information on Genbank's query fields here (http://www.ncbi.nlm.nih.gov/entrez/query/static/help/Summary_Matrices.html#Search_Fields_and_Qualifiers) .

That is how we would construct a query object, but we haven't retrieved sequences yet. To do so we will have to create a database object, some object that can get Sequence objects for us, just as we did in the first Genbank example:

```
use Bio::DB::GenBank;
use Bio::DB::Query::GenBank;

$query = "Arabidopsis[ORGN] AND topoisomerase[TITL] and 0:3000[SLEN]";
$query_obj = Bio::DB::Query::GenBank->new(-db    => 'nucleotide',  -query => $query );

$gb_obj = Bio::DB::GenBank->new;

$stream_obj = $gb_obj->get_Stream_by_query($query_obj);

while ($seq_obj = $stream_obj->next_seq) {
    # do something with the sequence object
    print $seq_obj->display_id, "\t", $seq_obj->length, "\n";
}
```

That `$stream_obj` and its `get_Stream_by_query` method may not look familiar. The idea is that you will use a stream whenever you expect to retrieve a stream or series of sequence objects. Much like `get_Seq_by_id`, but built to retrieve one or more objects, not just one object.

Notice how carefully separated the responsibilities of each object are in the code above: there's an object just to hold the query, an object to execute the query using this query object, an object to do the I/O, and finally the sequence object.

Warning. Be careful what you ask for, many of today's nucleotide database entries are genome-size and you

will probably run out of memory if your query happens to match one of these monstrosities. You can use the SLEN field to limit the size of the sequences you retrieve.

The Sequence Object

There's been a lot of discussion around the Sequence object, and this object has been created in a few different ways, but we haven't shown what it's capable of doing. The table below lists the methods available to you if you have a Sequence object in hand. "Returns" means what the object will give you when you ask it for data. Some methods, such as `seq()`, can be used to get or set values. You're setting when you assign a value, you're getting when you ask the object what values it has. For example, to get or retrieve a value

```
$sequence_as_string = $seq_obj->seq;
```

To set or assign a value:

```
$seq_obj->seq("MMTYDFFFFVVNNNNPPPPAAAW");
```

Table 1: Sequence Object Methods

Name	Returns	Example	Note
accession_number	identifier	$acc = $so->accession_number	get or set an identifier
alphabet	alphabet	$so->alphabet('dna')	get or set the alphabet ('dna','rna','protein')
authority	authority, if available	$so->authority("FlyBase")	get or set the organization
desc	description	$so->desc("Example 1")	get or set a description
display_id	identifier	$so->display_id("NP_123456")	get or set an identifier
division	division, if available (e.g. PRI)	$div = $so->division	get division (e.g. "PRI")
get_dates	array of dates, if available	@dates = $so->get_dates	get dates
get_secondary_accessions	array of secondary accessions, if available	@accs = $so->get_secondary_accessions	get other identifiers
is_circular	Boolean	if $so->is_circular { # }	get or set
keywords	keywords, if available	@array = $so->keywords	get or set keywords
length	length, a number	$len = $so->length	get the length
molecule	molecule type, if available	$type = $so->molecule	get molecule (e.g. "RNA", "DNA")

namespace	namespace, if available	$so->namespace("Private")	get or set the name space
new	Sequence object	$so = Bio::Seq->new(-seq => "MPQRAS")	create a new one, see Bio::Seq (http://search.cpan.org/search?query=Bio::Seq&mode=all) for more
pid	pid, if available	$pid = $so->pid	get pid
primary_id	identifier	$so->primary_id(12345)	get or set an identifier
revcom	Sequence object	$so2 = $so1->revcom	Reverse complement
seq	sequence string	$seq = $so->seq	get or set the sequence
seq_version	version, if available	$so->seq_version("1")	get or set a version
species	Species object	$species_obj = $so->species	See Bio::Species (http://search.cpan.org/search?query=Bio::Species&mode=all) for more
subseq	sequence string	$string = $seq_obj->subseq(10,40)	Arguments are start and end
translate	protein Sequence object	$prot_obj = $dna_obj->translate	
trunc	Sequence object	$so2 = $so1->trunc(10,40)	Arguments are start and end

The table above shows the methods you're likely to use that concern the Sequence object directly. Bear in mind that not all values, such as `molecule` or `division`, are found in all sequence formats, you have to know something about your input sequences in order to get some of these values.

There are also a number of methods that are concerned with the Features and Annotations associated with the Sequence object. This is something of a tangent but if you'd like to learn more see the Feature-Annotation HOWTO. The methods related to this topic are shown below.

Table 2: Feature and Annotation Methods

Name	Returns	Note
get_SeqFeatures	array of SeqFeature objects	
get_all_SeqFeatures	array of SeqFeature objects array	includes sub-features
remove_SeqFeatures	array of SeqFeatures removed	
feature_count	number of SeqFeature objects	
add_SeqFeature	annotation array of Annotation objects	get or set

Example Sequence Objects

Let's use some of the methods above and see what they return when the sequence object is obtained from different sources. In the Genbank example we're assuming we've used Genbank to retrieve or create a Sequence object. So this object could have have been retrieved like this:

```
use Bio::DB::GenBank;

$db_obj = Bio::DB::GenBank->new;
$seq_obj = $db_obj->get_Seq_by_acc("J01673");
```

Or it could have been created from a file like this:

```
use Bio::SeqIO;

$seqio_obj = Bio::SeqIO->new(-file => "J01673.gb", -format => "genbank" );
$seq_obj = $seqio_obj->next_seq;
```

What the Genbank file looks like:

```
LOCUS       ECORHO                  1880 bp    DNA     linear   BCT 26-APR-1993
DEFINITION  E.coli rho gene coding for transcription termination factor.
ACCESSION   J01673 J01674
VERSION     J01673.1  GI:147605
KEYWORDS    attenuator; leader peptide; rho gene; transcription terminator.
SOURCE      Escherichia coli
ORGANISM  Escherichia coli
                  Bacteria; Proteobacteria; Gammaproteobacteria; Enterobacteriales;
                  Enterobacteriaceae; Escherichia.
REFERENCE   1  (bases 1 to 1880)
AUTHORS   Brown,S., Albrechtsen,B., Pedersen,S. and Klemm,P.
TITLE     Localization and regulation of the structural gene for
             transcription-termination factor rho of Escherichia coli
JOURNAL   J. Mol. Biol. 162 (2), 283-298 (1982)
MEDLINE   83138788
PUBMED   6219230
REFERENCE   2  (bases 1 to 1880) AUTHORS   Pinkham,J.L. and Platt,T.
TITLE     The nucleotide sequence of the rho gene of E. coli K-12
JOURNAL   Nucleic Acids Res. 11 (11), 3531-3545 (1983)
MEDLINE   83220759
PUBMED   6304634
COMMENT      Original source text: Escherichia coli (strain K-12) DNA.
                    A clean copy of the sequence for [2] was kindly provided by
                    J.L.Pinkham and T.Platt.
FEATURES       Location/Qualifiers
     source      1..1880
                    /organism="Escherichia coli"
                    /mol_type="genomic DNA"
                    /strain="K-12"
                    /db_xref="taxon:562"
     mRNA       212..>1880
                    /product="rho mRNA"
     CDS          282..383
                    /note="rho operon leader peptide"
                    /codon_start=1
                    /transl_table=11
                    /protein_id="AAA24531.1"
                    /db_xref="GI:147606"
                    /translation="MRSEQISGSSLNPSCRFSSAYSPVTRQRKDMSR"
     gene         468..1727
                    /gene="rho"
     CDS          468..1727
                    /gene="rho"
                    /note="transcription termination factor"
                    /codon_start=1
                    /transl_table=11
                    /protein_id="AAA24532.1"
                    /db_xref="GI:147607"
                    /translation="MNLTELKNTPVSELITLGENMGLENLARMRKQDIIFAILKQHAK
                    SGEDIFGDGVLEILQDGFGFLRSADSSYLAGPDDIYVSPSQIRRFNLRTGDTISGKIR
```

```
                     PPKEGERYFALLKVNEVNFDKPENARNKILFENLTPLHANSRLRMERGNGSTEDLTAR
                     VLDLASPIGRGQRGLIVAPPKAGKTMLLQNIAQSIAYNHPDCVLMVLLIDERPEEVTE
                     MQRLVKGEVVASTFDEPASRHVQVAEMVIEKAKRLVEHKKDVIILLDSITRLARAYNT
                     VVPASGKVLTGGVDANALHRPKRFFGAARNVEEGGSLTIIATALIDTGSKMDEVIYEE
                     FKGTGNMELHLSRKIAEKRVFPAIDYNRSGTRKEELLTTQEELQKMWILRKIIHPMGE
                     IDAMEFLINKLAMTKTNDDFFEMMKRS"
ORIGIN      15 bp upstream from HhaI site.
        1 aaccctagca ctgcgccgaa atatggcatc cgtggtatcc cgactctgct gctgttcaaa
       61 aacggtgaag tggcggcaac caaagtgggt gcactgtcta aaggtcagtt gaaagagttc

                                  ...deleted...

     1801 tgggcatgtt aggaaaattc ctggaatttg ctggcatgtt atgcaatttg catatcaaat
     1861 ggttaatttt tgcacaggac
//
```

Either way, the values returned by various methods are shown below.

Table 3: Values from the Sequence object (Genbank)

Method	Returns
display_id	ECORHO
desc	E.coli rho gene coding for transcription termination factor.
display_name	ECORHO
accession	J01673
primary_id	147605
seq_version	1
keywords	attenuator; leader peptide; rho gene; transcription terminator
is_circular	
namespace	
authority	
length	1880
seq	AACCCT...ACAGGAC
division	BCT
molecule	DNA
get_dates	26-APR-1993
get_secondary_accessions	J01674

There's a few comments that need to be made. First, you noticed that there's an awful lot of information missing. All of this missing information is stored in what Bioperl calls Features and Annotations, see the Feature and Annotation HOWTO if you'd like to learn more about this. Second, a few of the methods don't return anything, like `namespace` and `authority`. The reason is that though these are good values in principle there are no commonly agreed upon standard names - perhaps someday the authors will be able to rewrite the code when all our public databases agree what these values should be. Finally, you may be wondering why the method names are what they are and why particular fields or identifiers end up associated with particular methods. Again, without having standard names for things that are agreed upon by the creators of our public databases all the authors could do is use common sense, and these choices seem to be reasonable ones.

Next let's take a look at the values returned by the methods used by the Sequence object when a fasta file is used as input. The fasta file entry looks like this, clearly much simpler than the corresponding Genbank entry:

```
>gi|147605|gb|J01673.1|ECORHO E.coli rho gene coding for transcription termination factor
AACCCTAGCACTGCGCCGAAATATGGCATCCGTGGTATCCCGACTCTGCTGCTGTTCAAAAACGGTGAAG
TGGCGGCAACCAAAGTGGGTGCACTGTCTAAAGGTCAGTTGAAAGAGTTCCTCGACGCTAACCTGGCGTA

                        ...deleted...

ACGTGTTTACGTGGCGTTTTGCTTTTATATCTGTAATCTTAATGCCGCGCTGGGCATGTTAGGAAAATTC
CTGGAATTTGCTGGCATGTTATGCAATTTGCATATCAAATGGTTAATTTTTGCACAGGAC
```

And here are the values:

Table 4: Values from the Sequence object (Fasta)

Method	Returns
display_id	147605|gb|J01673.1|ECORHO
desc	E.coli rho gene coding for transcription termination factor
display_name	147605|gb|J01673.1|ECORHO
accession	unknown
primary_id	147605|gb|J01673.1|ECORHO
is_circular	
namespace	
authority	
length	1880
seq	AACCCT...ACAGGAC

If you compare these values to the values taken from the Genbank entry you'll see that certain values are missing, like `seq_version`. That's because values like these aren't usually present in a fasta file.

Another natural question is why the values returned by methods like `display_id` are different even though the only thing distinguishing these entries are their respective formats. The reason is that there are no rules governing how one interconverts formats, meaning how Genbank creates fasta files from Genbank files may be different from how SwissProt performs the same interconversion. Until the organizations creating these databases agree on standard sets of names and formats all the Bioperl authors can do is do make reasonable choices.

Yes, Bioperl could follow the conventions of a single organization like Genbank such that `display_id` returns the same value when using Genbank format or Genbank's fasta format but the authors have elected not to base Bioperl around the conventions of any one organization.

Let's use a Swissprot file as our last example. The input entry looks like this:

```
ID   A2S3_RAT       STANDARD;      PRT;   913 AA.
AC   Q8R2H7; Q8R2H6; Q8R4G3;
DT   28-FEB-2003 (Rel. 41, Created)
DE   Amyotrophic lateral sclerosis 2 chromosomal region candidate gene
DE   protein 3 homolog (GABA-A receptor interacting factor-1) (GRIF-1) (O-
DE   GlcNAc transferase-interacting protein of 98 kDa).
GN   ALS2CR3 OR GRIF1 OR OIP98.
```

```
OS   Rattus norvegicus (Rat).
OC   Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi;
OC   Mammalia; Eutheria; Rodentia; Sciurognathi; Muridae; Murinae; Rattus.
OX   NCBI_TaxID=10116;
RN   [1]
RP   SEQUENCE FROM N.A. (ISOFORMS 1 AND 2), SUBCELLULAR LOCATION, AND
RP   INTERACTION WITH GABA-A RECEPTOR.
RC   TISSUE=Brain;
RX   MEDLINE=22162448; PubMed=12034717;
RA   Beck M., Brickley K., Wilkinson H.L., Sharma S., Smith M.,
RA   Chazot P.L., Pollard S., Stephenson F.A.;
RT   "Identification, molecular cloning, and characterization of a novel
RT   GABAA receptor-associated protein, GRIF-1.";
RL   J. Biol. Chem. 277:30079-30090(2002).
RN   [2]
RP   REVISIONS TO 579 AND 595-596, AND VARIANTS VAL-609 AND PRO-820.
RA   Stephenson F.A.;
RL   Submitted (FEB-2003) to the EMBL/GenBank/DDBJ databases.
RN   [3]
RP   SEQUENCE FROM N.A. (ISOFORM 3), INTERACTION WITH O-GLCNAC TRANSFERASE,
RP   AND O-GLYCOSYLATION.
RC   STRAIN=Sprague-Dawley; TISSUE=Brain;
RX   MEDLINE=22464403; PubMed=12435728;
RA   Iyer S.P.N., Akimoto Y., Hart G.W.;
RT   "Identification and cloning of a novel family of coiled-coil domain
RT   proteins that interact with O-GlcNAc transferase.";
RL   J. Biol. Chem. 278:5399-5409(2003).
CC   -!- SUBUNIT: Interacts with GABA-A receptor and O-GlcNac transferase.
CC   -!- SUBCELLULAR LOCATION: Cytoplasmic.
CC   -!- ALTERNATIVE PRODUCTS:
CC       Event=Alternative splicing; Named isoforms=3;
CC       Name=1; Synonyms=GRIF-1a;
CC         IsoId=Q8R2H7-1; Sequence=Displayed;
CC       Name=2; Synonyms=GRIF-1b;
CC         IsoId=Q8R2H7-2; Sequence=VSP_003786, VSP_003787;
CC       Name=3;
CC         IsoId=Q8R2H7-3; Sequence=VSP_003788;
CC   -!- PTM: O-glycosylated.
CC   -!- SIMILARITY: TO HUMAN OIP106.
DR   EMBL; AJ288898; CAC81785.2; -.
DR   EMBL; AJ288898; CAC81786.2; -.
DR   EMBL; AF474163; AAL84588.1; -.
DR   GO; GO:0005737; C:cytoplasm; IEP.
DR   GO; GO:0005634; C:nucleus; IDA.
DR   GO; GO:0005886; C:plasma membrane; IEP.
DR   GO; GO:0006357; P:regulation of transcription from Pol II pro...; IDA.
DR   InterPro; IPR006933; HAP1_N.
DR   Pfam; PF04849; HAP1_N; 1.
KW   Coiled coil; Alternative splicing; Polymorphism.
FT   DOMAIN      134    355       COILED COIL (POTENTIAL).
FT   VARSPLIC    653    672       VATSNPGKCLSFTNSTFTFT -> ALVSHHCPVEAVRAVHP
FT                                TRL (in isoform 2).
FT                                /FTId=VSP_003786.
FT   VARSPLIC    673    913       Missing (in isoform 2).
FT                                /FTId=VSP_003787.
FT   VARSPLIC    620    687       VQQPLQLEQKPAPPPPVTGIFLPPMTSAGGPVSVATSNPGK
FT                                CLSFTNSTFTFTTCRILHPSDITQVTP -> GSAASSTGAE
FT                                ACTTPASNGYLPAAHDLSRGTSL (in isoform 3).
FT                                /FTId=VSP_003788.
FT   VARIANT     609    609       E -> V.
FT   VARIANT     820    820       S -> P.
SQ   SEQUENCE   913 AA;  101638 MW;  D0E135DBEC30C28C CRC64;
     MSLSQNAIFK SQTGEENLMS SNHRDSESIT DVCSNEDLPE VELVNLLEEQ LPQYKLRVDS
     LFLYENQDWS QSSHQQQDAS ETLSPVLAEE TFRYMILGTD RVEQMTKTYN DIDMVTHLLA
                              ...deleted...
     GIARVVKTPV PRENGKSREA EMGLQKPDSA VYLNSGGSLL GGLRRNQSLP VMMGSFGAPV
     CTTSPKMGIL KED
//
```

The corresponding set of values is shown below.

Table 5: Values from the Sequence object (Swissprot)

Method	Returns

display_id	A2S3_RAT
desc	Amyotrophic lateral ... protein of 98 kDa).
display_name	A2S3_RAT
accession	Q8R2H7
is_circular	
namespace	
authority	
seq_version	
keywords	Coiled coil; Alternative splicing; Polymorphism
length	913
seq	MSLSQ...ILKED
division	RAT
get_dates	28-FEB-2003 (Rel. 41, Created)
get_secondary_accessions	Q8R2H6 Q8R4G3

As in the Genbank example there's information that the Sequence object doesn't supply, and it's all stored in Annotation objects. See the Feature and Annotation HOWTO for more.

Translating

Translation in bioinformatics can mean slightly different things, either translating a nucleotide sequence from start to end or translate the actual coding regions in mRNAs or cDNAs. The Bioperl implementation of sequence translation does both of these.

Any sequence object with alphabet 'dna' or 'rna' can be translated by simply using `translate` which returns a protein sequence object:

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

All codons will be translated, including those before and after any initiation and termination codons. For example, **ttttttatgccctaggggg** will be translated to **FFMP*G**

However, the `translate()` method can also be passed several optional parameters to modify its behavior. For example, you can tell `translate()` to modify the characters used to represent terminator (default is *****) and unknown amino acids (default is **X**).

```
$prot_obj = $my_seq_object->translate(-terminator => '-');
$prot_obj = $my_seq_object->translate(-unknown => '_');
```

You can also determine the frame of the translation. The default frame starts at the first nucleotide (frame 0). To get translation in the next frame we would write:

```
$prot_obj = $my_seq_object->translate(-frame => 1);
```

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 checks. Specifically, `translate()` needs to confirm that the open reading frame has appropriate start and terminator codons at the very beginning and the very end of the sequence and that there are no terminator codons present within the sequence in frame 0. 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 "complete" to 1:

```
$prot_obj = $my_seq_object->translate(-complete => 1);
```

If "complete" is set to true and the criteria for a proper CDS are not met, the method, by default, issues a warning. By setting "throw" to 1, one can instead instruct the program to die if an improper CDS is found, e.g.

```
$prot_obj = $my_seq_object->translate(-complete => 1,
                                      -throw => 1);
```

The codontable_id argument to `translate()` makes it possible to use alternative genetic codes. There are currently 16 codon tables defined, including 'Standard', 'Vertebrate Mitochondrial', 'Bacterial', 'Alternative Yeast Nuclear' and 'Ciliate, Dasycladacean and Hexamita Nuclear'. All these tables can be seen in Bio::Tools::CodonTable (http://search.cpan.org/search?query=Bio::Tools::CodonTable&mode=all) . For example, for mitochondrial translation:

```
$prot_obj = $seq_obj->translate(-codontable_id => 2);
```

You can also create a custom codon table and pass this to `translate`, the code will look something like this:

```
use Bio::Tools::CodonTable;

@custom_table =
    ( 'test1',
      'FFLLSSSSYY**CC*WLLLL**PPHHQQR*RRIIIFT*TT*NKKSSRRV*VVAA*ADDEE*GGG'
    );

$codon_table = Bio::Tools::CodonTable->new;

$id = $codon_table->add_table(@custom_table);

$prot_obj = $my_seq_object->translate(-codontable_id => $id);
```

See Bio::Tools::CodonTable (http://search.cpan.org/search?query=Bio::Tools::CodonTable&mode=all) for information on the format of a codon table.

`translate()` can also find the open reading frame (ORF) starting at the 1st initiation codon in the nucleotide sequence, regardless of its frame, and translate that:

```
$prot_obj = $my_seq_object->translate(-orf => 1);
```

Most of the codon tables, including the default codon table NCBI "Standard", have initiation codons in addition to ATG. To tell `translate()` to use only ATG or atg as the initiation codon set -start to "atg":

```
$prot_obj = $my_seq_object->translate(-orf => 1,
                                      -start => "atg" );
```

The -start argument only applies when -orf is set to 1.

Last trick. By default `translate()` will translate the termination codon to some special character (the default is *, but this can be reset using the -terminator argument).

When -complete is set to 1 this character is removed. So, with this:

```
$prot_obj = $my_seq_object->translate(-orf => 1,
                                      -complete => 1);
```

the sequence **tttttatgccctaggggg** will be translated to **MP**, not **MP***.

See Bio::Tools::CodonTable (http://search.cpan.org/search?query=Bio::Tools::CodonTable&mode=all) and Bio::PrimarySeqI (http://search.cpan.org/search?query=Bio::PrimarySeqI&mode=all) for more information on translation.

Obtaining basic sequence statistics

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, Bio::Tools::SeqStats (http://search.cpan.org/search?query=Bio::Tools::SeqStats&mode=all) 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, Bio::Tools::SeqStats (http://search.cpan.org/search?query=Bio::Tools::SeqStats&mode=all) also returns counts of the number of codons used. For example:

```
use Bio::Tools::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" (e.g. tetramers or hexamers) within the sequence. See Bio::Tools::SeqStats (http://search.cpan.org/search?query=Bio::Tools::SeqStats&mode=all) and Bio::Tools::SeqWords (http://search.cpan.org/search?query=Bio::Tools::SeqWords&mode=all) for more information.

BLAST

This section is outdated, please see HOWTO:BlastPlus. BLAST is no longer supported by NCBI, it has been superceded by BLAST+.

You have access to a large number of sequence analysis programs within Bioperl. Typically this means you have a means to run the program and frequently a means of parsing the resulting output, or report, as well. Certainly the most popular analytical program is BLAST so let's use it as an example. First you'll need to get BLAST (http://www.ncbi.nlm.nih.gov/blast/) , also known as blastall, installed on your machine and running, versions of the program that can run on all the popular operating systems can be downloaded (http://www.ncbi.nlm.nih.gov/blast/download.shtml) from NCBI. The example code assumes that you used the formatdb program to index the database sequence file "db.fa".

As usual, we start by choosing a module to use, in this case Bio::Tools::Run::StandAloneBlast

(http://search.cpan.org/search?query=Bio::Tools::Run::StandAloneBlast&mode=all) . You stipulate some blastall parameters used by the blastall program by using `new()`. As you'd expect, we want to create a Blast object, and we will pass a Sequence object to the Blast object, this Sequence object will be used as the query:

```
use Bio::Seq;
use Bio::Tools::Run::StandAloneBlast;

$blast_obj = Bio::Tools::Run::StandAloneBlast->new(-program  => 'blastn', -database => 'db.fa');

$seq_obj = Bio::Seq->new(-id  =>"test query", -seq =>"TTTAAATATATTTTGAAGTATAGATTATATGTT");

$report_obj = $blast_obj->blastall($seq_obj);

$result_obj = $report_obj->next_result;

print $result_obj->num_hits;
```

By calling the `blastall` method you're actually running BLAST, creating the report file, and parsing the report file's contents. All the data in the report ends up in the report object, and you can access or print out the data in all sorts of ways. The report object, `$report_obj`, and the result object, `$result_obj`, come from the SearchIO modules. The SearchIO HOWTO will tell you all about using these objects to extract useful data from your BLAST analyses.

Here's an example of how one would use SearchIO to extract data from a BLAST report:

```
use Bio::SearchIO;
$report_obj = new Bio::SearchIO(-format => 'blast',
                                -file   => 'report.bls');
while( $result = $report_obj->next_result ) {
    while( $hit = $result->next_hit ) {
      while( $hsp = $hit->next_hsp ) {
         if ( $hsp->percent_identity > 75 ) {
           print "Hit\t", $hit->name, "\n", "Length\t", $hsp->length('total'),
                   "\n", "Percent_id\t", $hsp->percent_identity, "\n";
         }
       }
     }
}
```

This code prints out details about the match when the HSP or aligned pair are greater than 75% identical.

Sometimes you'll see errors when you try to use Bio::Tools::Run::StandAloneBlast (http://search.cpan.org/search?query=Bio::Tools::Run::StandAloneBlast&mode=all) that have nothing to do with Bioperl. Make sure that BLAST is set up properly and running before you attempt to script it using Bio::Tools::Run::StandAloneBlast (http://search.cpan.org/search?query=Bio::Tools::Run::StandAloneBlast&mode=all) . There are some notes on setting up BLAST in the INSTALL (http://bioperl.open-bio.org/SRC/bioperl-live/INSTALL) file.

Bioperl enables you to run a wide variety of bioinformatics programs but in order to do so, in most cases, you will need to install the accessory bioperl-run package. In addition there is no guarantee that there is a corresponding parser for the program that you wish to run, but parsers have been built for the most popular programs. You can find the bioperl-run package on the download page.

Indexing for Fast Retrieval

One of the under-appreciated features of Bioperl is its ability to index sequence files. The idea is that you would create some sequence file locally and create an index file for it that enables you to retrieve sequences from the sequence file. Why would you want to do this? Speed, for one. Retrieving sequences from local,

indexed sequence files is much faster than using the Bio::DB::GenBank (http://search.cpan.org/search?query=Bio::DB::GenBank&mode=all) module used above that retrieves from a remote database. It's also much faster than using SeqIO, in part because SeqIO is stepping through a file, one sequence at a time, starting at the beginning of the file. Flexibility is another reason. What if you'd created your own collection of sequences, not found in a public database? By indexing this collection you'll get fast access to your sequences.

There's only one requirement here, the term or id that you use to retrieve the sequence object must be unique in the index, these indices are not built to retrieve multiple sequence objects from one query.

There are a few different modules in Bioperl that can index sequence files, the Bio::Index::* modules and Bio::DB::Fasta (http://search.cpan.org/search?query=Bio::DB::Fasta&mode=all) . All these modules are scripted in a similar way: you first index one or more files, then retrieve sequences from the indices. Let's begin our script with the use statement and also set up our environment with some variables (the sequence file, FASTA sequence format, will be called "sequence.fa"):

```
use Bio::Index::Fasta;
$ENV{BIOPERL_INDEX_TYPE} = "SDBM_File";
```

The lines above show that you can set environmental variables from within Perl and they are stored in Perl's own `%ENV` hash. This is essentially the same thing as the following in tcsh or csh:

```
>setenv BIOPERL_INDEX_TYPE SDBM_File
```

Or the following in the bash shell:

```
>export BIOPERL_INDEX_TYPE=SDBM_File
```

The BIOPERL_INDEX_TYPE variable refers to the indexing scheme, and SDBM_File is the scheme that comes with Perl. BIOPERL_INDEX stipulates the location of the index file, and this way you could have more than one index file per sequence file if you wanted, by designating multiple locations (and the utility of more than 1 index will become apparent).

Now let's construct the index:

```
$ENV{BIOPERL_INDEX_TYPE} = "SDBM_File";
use Bio::Index::Fasta;

$file_name = "sequence.fa";
$id = "48882";
$inx = Bio::Index::Fasta->new (-filename => $file_name . ".idx", -write_flag => 1);
$inx->make_index($file_name);
```

You would execute this script in the directory containing the "sequence.fa" file, and it would create an index file called "sequence.fa.idx". Then you would retrieve a sequence object like this:

```
$seq_obj = $inx->fetch($id)
```

By default the fasta indexing code will use the string following the **>** character as a key, meaning that fasta header line should look something like this if you want to fetch using the value "48882":

```
>48882 pdb|1CRA
```

However, what if you wanted to retrieve using some other key, like "1CRA" in the example above? You can customize the index by using Bio::Index::Fasta (http://search.cpan.org/search?query=Bio::Index::Fasta&mode=all) 's `id_parser` method, which accepts the name of a function as an argument where that function tells the indexing object what key to use. For example:

```
$inx->id_parser(\&get_id);
$inx->make_index($file_name);

sub get_id {
    $header = shift;
    $header =~ /pdb\|(\S+)/;
    $1;
}
```

To be precise, one would say that the `id_parser` method accepts a reference to a function as an argument.

Bio::DB::Fasta (http://search.cpan.org/search?query=Bio::DB::Fasta&mode=all) has some features that Bio::Index::Fasta (http://search.cpan.org/search?query=Bio::Index::Fasta&mode=all) lacks, one of the more useful ones is that it was built to handle very large sequences and can retrieve sub-sequences from genome-size sequences efficiently. Here is an example:

```
use Bio::DB::Fasta;

($file,$id,$start,$end) = ("genome.fa","CHROMOSOME_I",11250,11333);

$db = Bio::DB::Fasta->new($file);

$seq = $db->seq($id,$start,$end);

print $seq,"\n";
```

This script indexes the genome.fa file, then retrieves a sub-sequence of CHROMOSOME_I, starting at 11250 and ending at 11333. One can also specify what ids can be used as keys, just as in Bio::Index::Fasta (http://search.cpan.org/search?query=Bio::Index::Fasta&mode=all) .

There's a bit more information on indexing in HOWTO:Local_Databases.

Searching for genes in genomic DNA

Parsers for widely used gene prediction programs - Genscan, Sim4, Genemark, Grail, ESTScan and MZEF - are available in Bioperl. The interfaces for these parsers are all similar. The syntax is relatively self-explanatory, see Bio::Tools::Genscan (http://search.cpan.org/search?query=Bio::Tools::Genscan&mode=all) , Bio::Tools::Genemark (http://search.cpan.org/search?query=Bio::Tools::Genemark&mode=all) , Bio::Tools::Grail (http://search.cpan.org/search?query=Bio::Tools::Grail&mode=all) , Bio::Tools::ESTScan (http://search.cpan.org/search?query=Bio::Tools::ESTScan&mode=all) , Bio::Tools::MZEF (http://search.cpan.org/search?query=Bio::Tools::MZEF&mode=all) , and Bio::Tools::Sim4::Results (http://search.cpan.org/search?query=Bio::Tools::Sim4::Results&mode=all) for further details. Here are some examples on how to use these modules.

```
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::Tools::Prediction::Gene (http://search.cpan.org/search?query=Bio::Tools::Prediction::Gene&mode=all) and Bio::Tools::Prediction::Exon (http://search.cpan.org/search?query=Bio::Tools::Prediction::Exon&mode=all) 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::SeqFeature::Generic (http://search.cpan.org/search?query=Bio::SeqFeature::Generic&mode=all) and Bio::Tools::Sim4::Exons (http://search.cpan.org/search?query=Bio::Tools::Sim4::Exons&mode=all) 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::Tools::EPCR (http://search.cpan.org/search?query=Bio::Tools::EPCR&mode=all) . 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);
}
```

Code to query bibliographic databases

Bio::Biblio (http://search.cpan.org/search?query=Bio::Biblio&mode=all) objects are used to query bibliographic databases, such as MEDLINE. The associated modules are built to work with OpenBQS-compatible databases (see http://www.ebi.ac.uk/~senger/openbqs). A Bio::Biblio (http://search.cpan.org/search?query=Bio::Biblio&mode=all) object can execute a query like:

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

See Bio::Biblio (http://search.cpan.org/search?query=Bio::Biblio&mode=all) , the scripts/biblio/biblio.PLS script, or the examples/biblio/biblio_examples.pl script for more information.

Using EMBOSS applications with Bioperl

EMBOSS (European Molecular Biology Open Source Software) is an extensive collection of sequence analysis programs written in the C programming language (http://emboss.sourceforge.net/). There are a number of algorithms in EMBOSS that are not found in Bioperl (e.g. calculating DNA melting temperature, finding repeats, identifying prospective antigenic sites) so if you cannot find the function you want in Bioperl you might be able to find it in EMBOSS. The Bioperl code that runs EMBOSS programs is Bio::Factory::EMBOSS (http://search.cpan.org/search?query=Bio::Factory::EMBOSS&mode=all) .

EMBOSS programs are usually called from the command line but the bioperl-run auxiliary library provides a Perl wrapper for EMBOSS function calls so that they can be executed from within a Perl script. Of course, the EMBOSS package as well as the bioperl-run must be installed in order for the Bioperl wrapper to function.

An example of the Bioperl Bio::Factory::EMBOSS (http://search.cpan.org /search?query=Bio::Factory::EMBOSS&mode=all) wrapper where a file is returned would be:

```
use Bio::Factory::EMBOSS;

$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. It's worth mentioning that another way to align sequences in Bioperl is to run a program from the EMBOSS suite, such as 'matcher'. This can produce an output file that Bioperl can read in using AlignIO:

```
my $factory = new Bio::Factory::EMBOSS;
my $prog = $factory->program('matcher');

$prog->run({ -sequencea => Bio::Seq->new(-id => "seq1",
                                         -seq => $seqstr1),
             -sequenceb => Bio::Seq->new(-id => "seq2",
                                         -seq => $seqstr2),
             -aformat      => "pair",
             -alternatives => 2,
             -outfile     => $outfile});

my $alignio_fmt = "emboss";
my $align_io = new Bio::AlignIO(-format => $alignio_fmt,
                                -file   => $outfile);
```

More on Bioperl

Perhaps this article has gotten you interested in learning a bit more about Bioperl. Here are some other things you might want to look at:

- HOWTOs. Each one covers a topic in some detail, but there are certainly some HOWTOs that are missing that we would like to see written. Would you like to become an expert and write one yourself?
- The module documentation. Each module is documented, but the quality and quantity varies by

module.

- Bioperl scripts. You'll find them in the scripts/ directory and in the examples/ directory of the Bioperl package. The former contains more carefully written and documented scripts that can be installed along with Bioperl. You should feel free to contribute scripts to either of these directories.

Perl's Documentation System

The documentation for Perl is available using a system known as POD (http://perldoc.perl.org/perlpod.html) , which stands for Plain Old Documentation. You can access this built-in documentation by using the `perldoc` command. To view information on how to use `perldoc`, type the following at the command line:

```
>perldoc perldoc
```

Perldoc is a very useful and versatile tool, shown below are some more examples on how to use perldoc. Read about Perl's built-in `print` function:

```
>perldoc -f print
```

Read about any module, including any of the Bioperl modules:

```
>perldoc Bio::SeqIO
```

The Basics of Perl Objects

Object-oriented programming (OOP) is a software engineering technique for modularizing code. The difference between object-oriented programming and procedural programming can be simply illustrated.

A Simple Procedural Example

Assume that we have a DNA sequence stored in the scalar variable `$sequence`. We'd like to generate the reverse complement of this sequence and store it in `$reverse_complement`. Shown below is the procedural Perl technique of using a function, or sub-routine, to operate on this scalar data:

```
use Bio::Perl;

$reverse_complement = revcom( $sequence );
```

The hallmark of a procedural program is that data and functions to operate on that data are kept separate. In order to generate the reverse complement of a DNA sequence, we need to call a function that operates on that DNA sequence.

A Simple Object-Oriented Example

Shown below is the object-oriented way of generating the reverse complement of a DNA sequence:

```
$reversed_obj = $seq_obj->revcom;
```

The main difference between this object-oriented example and the procedural example shown before is that the method for generating the reverse complement, `revcom`, is part of `$seq_obj`. To put it another way, the object `$seq_obj` knows how to calculate and return its reverse complement. Encapsaluting both data and functions into the same construct is the fundamental idea behind object-oriented programming.

Terminology

In the object-oriented example above, `$seq_obj` is called an object, and `revcom` is called a method. An object is a data structure that has both data and methods associated with it. Objects are separated into types called classes, and the class of an object defines both the data that it can hold and the methods that it knows. A specific object that has a defined class is referred to as an instance of that class. In Perl you could say that each module is actually a class, but for some reason the author of Perl elected to use the term "module" rather than "class".

That's the sort of explanation you'll get in most programming books, but what is a Perl object really? Usually a hash. In Bioperl the data that the object contains is stored in a single, complex hash and the object, like `$seq_obj`, is a reference to this hash. In addition, the methods that the object can use are also stored in this hash as particular kinds of references. You could say that an object in Perl is a special kind of hash reference.

Bioperl uses the object-oriented paradigm, and here are some texts if you want to learn more:

- Object Oriented Perl (http://www.manning.com/conway/)
- Bioperl's Developer Information, particularly the Advanced BioPerl page, for anyone who'd like to write their own modules.
- The ENSEMBL Perl API (http://www.ensembl.org/info/docs/api/core/core_tutorial.html) , a way of accessing ENSEMBL (http://www.ensembl.org) 's genomics data in a manner very much like Bioperl.