Bio::Graphics HOWTO

Lincoln Stein

Cold Spring Harbor Laboratory [http://www.cshl.org]

<lstein@cshl.org>

2002-09-01

Revision History		
Revision 0.2	2003-05-15	lds
Current as of BioPerl 1.2.2		
Revision History		
Revision 0.3	2003-09-17	BIO
Current as of BioPerl 1.2.3		

This HOWTO describes how to render sequence data graphically in a horizontal map. It applies to a variety of situations ranging from rendering the feature table of a GenBank entry, to graphing the positions and scores of a BLAST search, to rendering a clone map. It describes the programmatic interface to the Bio::Graphics module, and discusses how to create dynamic web pages using Bio::DB::GFF and the gbrowse package.

Table of Contents

1. Introduction

This HOWTO describes the Bio::Graphics module, and some of the applications that were built on top of it. Bio::Graphics was designed to solve the following common problems:

- You have a list of BLAST hits on a sequence and you want to generate a picture that shows where the hits go and what their score is.
- You have a big GenBank file with a complex feature table, and you want to render the positions of the genes, repeats, promoters and other features.
- You have a list of ESTs that you've mapped to a genome, and you want to show how they align.

- You have created a clone fingerprint map, and you want to display it.

The Bio::Graphics module was designed to solve these problems. In addition, using the Bio::DB::GFF module (part of BioPerl) and the gbrowse program (available from http://www.gmod.org) you can create interactive web pages to explore your data.

This document takes you through a few common applications of Bio::Graphics in a cookbook fashion.

2. Preliminaries

Bio::Graphics is dependent on GD, a Perl module for generating bitmapped graphics written by the author. GD in turn is dependent on libgd, a C library written by Thomas Boutell, formerly also of Cold Spring Harbor Laboratory (www.boutell.com/gd). To use Bio::Graphics, you must have both these software libraries installed.

If you are on a Linux system, you might already have GD installed. To verify, run the following command:

```
 % perl -MGD -e 'print $GD::VERSION';
```

If the program prints out a version number, you are in luck. Otherwise, if you get a "Can't locate GD.pm" error, you'll have to install the module. For users of ActiveState Perl this is very easy. Just start up the PPM program and issue the command "install GD". For users of other versions of Perl, you should go to www.cpan.org, download a recent version of the GD module, unpack it, and follow the installation directions. You may also need to upgrade to a recent version of the libgd C library.

If the program prints out a version number, you are in luck. Otherwise, if you get a "Can't locate GD.pm" error, you'll have to install the module. For users of ActiveState Perl this is very easy. Just start up the PPM program and issue the command "install GD". For users of other versions of Perl, you should go to www.cpan.org, download a recent version of the GD module, unpack it, and follow the installation directions.

You may need to upgrade to a recent version of the libgd C library. At the time this was written, there were two versions of libgd. libgd version 1.8.4 is the stable version, and corresponds to GD version 1.43. libgd version 2.0.1 is the beta version; although it has many cool features, it also has a few known bugs (which Bio::Graphics works around). If you use libgd 2.0.1 or higher, be sure it matches GD version 2.0.1 or higher.

You will also need to install the Text::Shellwords module, which is available from CPAN.

3. Getting Started

All the code examples and BLAST input files we'll use are available in the doc/howto/examples/graphics directory in the BioPerl package.

Our first example will be rendering a table of BLAST hits on a sequence that is exactly 1000 residues long. For now, we're ignoring finicky little details like HSPs, and assume that each hit is a single span from start to end. Also, we'll be using the BLAST score rather than P or E value. Later on, we'll switch to using real BLAST output parsed by the Bio::SearchIO module, but for now, our table looks like this:

```
# hit          score start end
hsHOX3          381 2 200
scHOX3          210 2 210
xlHOX3          800 2 200
hsHOX2          1000 380 921
scHOX2          812 402 972
xlHOX2          1200 400 970
BUM          400 300 620
```

```
PRES1         127 310 700
```

Figure 1. Simple blast hit file (data1.txt)

Our first attempt to parse and render this file looks like this:

Example 1. Rendering the simple blast hit file (render_blast1.pl)

```
 0  #!/usr/bin/perl

 1  # This is code example 1 in the Graphics-HOWTO
 2  use strict;
 3  use Bio::Graphics;
 4  use Bio::SeqFeature::Generic;

 5  my $panel = Bio::Graphics::Panel->new(-length => 1000,-width  => 800);
 6  my $track = $panel->add_track(-glyph => 'generic',-label  => 1);

 7  while (<>) { # read blast file
 8    chomp;
 9    next if /^\#/;  # ignore comments
10    my($name,$score,$start,$end) = split /\t+/;
11    my $feature = Bio::SeqFeature::Generic->new(-display_name=>$name,-score=>$score,
12                                                -start=>$start,-end=>$end);
13    $track->add_feature($feature);
14  }

15  print $panel->png;
```

The script begins by loading the Bio::Graphics module (line 3), which in turn brings in a number of other modules that we'll use later. We also load `Bio::SeqFeature::Generic` in order to create a series of Bio::SeqFeatureI objects for rendering. We then create a Bio::Graphics::Panel object by calling its new() method, specifying that the panel is to correspond to a sequence that is 1000 nucleotides long, and has a physical width of 800 pixels (line 5). The Panel can contain multiple horizontal tracks, each of which has its own way of rendering features (called a "glyph"), color, labeling convention, and so forth. In this simple example, we create a single track by calling the panel object's add_track() method (line 6), specify a glyph type of "generic", and ask that the objects in the track be labeled by providing a true value to the -label argument. This gives us a track object that we can add our hits to.

We're now ready to render the blast hit file. We loop through it (line 7-14), stripping off the comments, and parsing out the name, score and range (line 10). We now need a Bio::SeqFeatureI object to place in the track. The easiest way to do this is to create a Bio::SeqFeature::Generic object, which is similar to Bio::PrimarySeq, except that it provides a way of attaching start and end positions to the sequence, as well as such nebulous but useful attributes as the "score" and "source". The Bio::SeqFeature::Generic->new() method, invoked in line 11, takes arguments corresponding to the name of each hit, its start and end coordinates, and its score.

After creating the feature object, we add it to the track by calling the track's add_feature() method (line 13).

After processing all the hits, we call the panel's png() method to render them and convert it into a Portable Network Graphics file, the contents of which are printed to standard output. We can now view the result by piping it to our favorite image display program.

IMPORTANT NOTE: If you are on a Windows platform, you need to put STDOUT into binary mode so that the PNG file does not go through Window's carriage return/linefeed transformations. Before the final print statement, put the statement "binmode(STDOUT)".

This advice also applies to certain versions of RedHat, which ship with a patched (and possibly broken) version of Perl.

```
% render1.pl data1.txt | display -
```

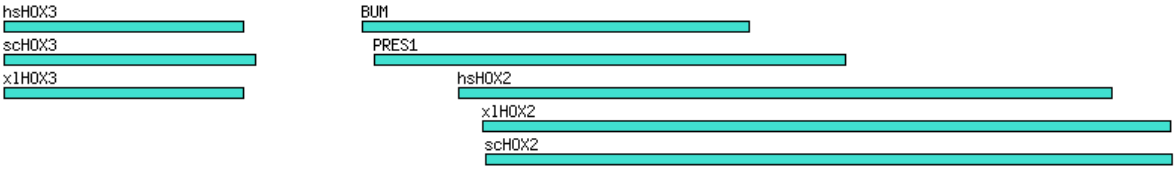

Figure 2. Rendering BLAST hits

Users of operating systems that don't support pipes can simply redirect the output to a file and view it in their favorite image program.

4. Adding a Scale to the Image

This is all very nice, but it's missing two essential components:

- It doesn't have a scale.
- It doesn't distinguish between hits with different scores.

Example 2 fixes these problems

Example 2. Rendering the blast hit file with scores and scale

```
 0  #!/usr/bin/perl

 1  # This is code example 2 in the Graphics-HOWTO
 2  use strict;
 3  use lib '/home/lstein/projects/bioperl-live';
 4  use Bio::Graphics;
 5  use Bio::SeqFeature::Generic;

 6  my $panel = Bio::Graphics::Panel->new(-length => 1000,
 7                                        -width  => 800,
 8                                        -pad_left => 10,
 9                                        -pad_right => 10,
10                                       );
11  my $full_length = Bio::SeqFeature::Generic->new(-start=>1,-end=>1000);
12  $panel->add_track($full_length,
13                    -glyph   => 'arrow',
14                    -tick    => 2,
15                    -fgcolor => 'black',
16                    -double  => 1,
17                   );

18  my $track = $panel->add_track(-glyph => 'graded_segments',
19                                -label  => 1,
20                                -bgcolor => 'blue',
21                                -min_score => 0,
22                                -max_score => 1000);

23  while (<>) { # read blast file
24    chomp;
```

```
25    next if /^\#/;  # ignore comments
26    my($name,$score,$start,$end) = split /\t+/;
27    my $feature = Bio::SeqFeature::Generic->new(-display_name=>$name,-score=>$score,
28                                                -start=>$start,-end=>$end);
29    $track->add_feature($feature);
30  }

31  print $panel->png;
```

There are several changes to look at. The first is minor. We'd like to put a boundary around the left and right edges of the image so that the features don't bump up against the margin, so we specify a 10 pixel leeway with the *-pad_left* and *-pad_right* arguments in lines 8 and 9.

The next change is more subtle. We want to draw a scale all the way across the image. To do this, we create a track to contain the scale, and a feature that spans the track from the start to the end. Line 11 creates the feature, giving its start and end coordinates. Lines 12-17 create a new track containing this feature. Unlike the previous example, in which we created the track first and then added features one at a time with add_feature(), we show here how to add feature(s) directly in the call to add_track(). If the first argument to add_track is either a single feature or a feature array ref, then add_track() will automatically incorporate the feature(s) into the track in a single efficient step. The remainder of the arguments configure the track as before. The -glyph argument says to use the "arrow" glyph. The -tick argument indicates that the arrow should contain tick marks, and that both major and minor ticks should be shown (tick type of "2"). We set the foreground color to black, and request that arrows should be placed at both ends (-double =>1).

1

In lines 18-22, we get a bit fancier with the blast hit track. Now, instead of creating a generic glyph, we use the "graded_segments" glyph. This glyph takes the specified background color for the feature, and either darkens or lightens it according to its score. We specify the base background color (-bgcolor => 'blue'), and the minimum and maximum scores to scale to (-min_score and -max_score). (You may need to experiment with the min and max scores in order to get the glyph to scale the colors the way you want.) The remainder of the program is the same.

When we run the modified script, we get this image.

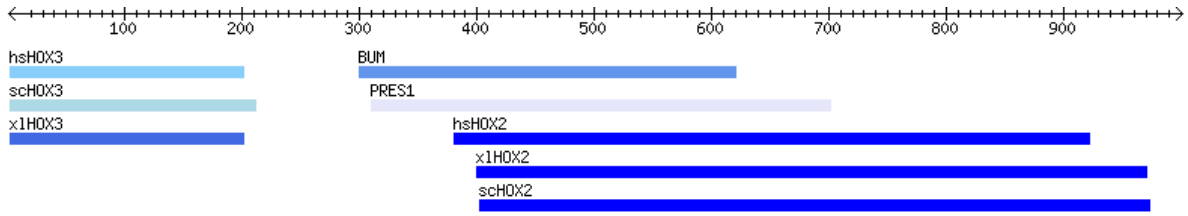

Figure 3. The improved image

IMPORTANT NOTE: Remember that if you are on a Windows platform, you need to put STDOUT into binary mode so that the PNG file does not go through Window's carriage return/linefeed transformations. Before the final print statement, write binmode(STDOUT).

5. Improving the Image

1Obtain the list of glyphs by running perldoc on Bio::Graphics::Glyph. Obtain a description of the glyph options by running perldoc on individual glyphs, for example "perldoc Bio::Graphics::Glyph::arrow."

Before we move into displaying gapped alignments, let's tweak the image slightly so that higher scoring hits appear at the top of the image, and the score itself is printed in red underneath each hit. The changes are shown in Example 3.

Example 3. Rendering the blast hit file with scores and scale

```
 0  #!/usr/bin/perl

 1  # This is code example 3 in the Graphics-HOWTO
 2  use strict;
 3  use lib '/home/lstein/projects/bioperl-live';
 4  use Bio::Graphics;
 5  use Bio::SeqFeature::Generic;

 6  my $panel = Bio::Graphics::Panel->new(-length => 1000,
 7                                        -width  => 800,
 8                                        -pad_left => 10,
 9                                        -pad_right => 10,
10                                       );
11  my $full_length = Bio::SeqFeature::Generic->new(-start=>1,-end=>1000);
12  $panel->add_track($full_length,
13                    -glyph   => 'arrow',
14                    -tick    => 2,
15                    -fgcolor => 'black',
16                    -double  => 1,
17                   );

18  my $track = $panel->add_track(-glyph => 'graded_segments',
19                                -label  => 1,
20                                -bgcolor => 'blue',
21                                -min_score => 0,
22                                -max_score => 1000,
23                                -font2color     => 'red',
24                                -sort_order     => 'high_score',
25                                -description => sub {
26                                  my $feature = shift;
27                                  my $score   = $feature->score;
28                                  return "score=$score";
29                                 });

30  while (<>) { # read blast file
31    chomp;
32    next if /^\#/;  # ignore comments
33    my($name,$score,$start,$end) = split /\t+/;
34    my $feature = Bio::SeqFeature::Generic->new(-score=>$score,
35                                                -display_name=>$name,
36                                                -start=>$start,-end=>$end);
37    $track->add_feature($feature);
38  }

39  print $panel->png;
```

There are two changes to look at. The first appears in line 24, where we pass the *-sort_order* option to the call that creates the blast hit track. *-sort_order* changes the way that features sort from top to bottom, and will accept a number of prepackaged sort orders or a coderef for custom sorting. In this case, we pass a prepackaged sort order of *high_score*, which sorts the hits from top to bottom in reverse order of their score.

The second change is more complicated, and involves the -description option that appears in the `add_track()` call on lines 25-28. The value of *-description* will be printed beneath each feature. We could pass *-description* a constant string, but that would simply print the same string under each feature. Instead we pass *-description* a code reference to a subroutine that will be invoked while the picture is being rendered. This subroutine will be passed the current feature, and must return the string to use as the value of the description. In our code, we simply fetch out the BLAST hit's score using its `score()` method, and incorporate that into the description string.

Tip

The ability to use a code reference as a configuration option isn't unique to *-description*. In fact, you can use a code reference for any of the options passed to add_track().

Another minor change is the use of *-font2color* in line 23. This simply sets the color used for the description strings. Figure 3 shows the effect of these changes.

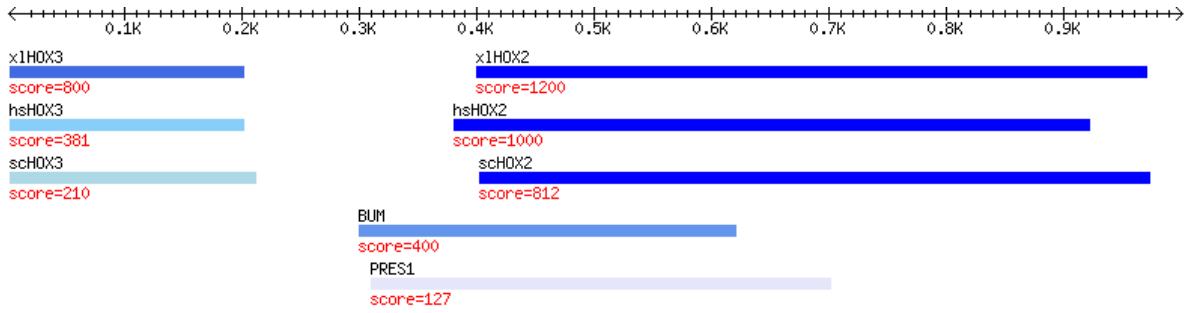

Figure 4. The image with descriptions and sorted hits

6. Parsing Real BLAST Output

From here it's just a small step to writing a general purpose utility that will read a BLAST file, parse its output, and output a picture. The key is to use the `Bio::SearchIO` infrastructure because it produces Bio::SeqFeatureI similarity hits that can be rendered directly by `Bio::Graphics`.

Code example 4 shows the new utility.

Example 4. Parsing and Rendering a Real BLAST File with Bio::SearchIO

```
 0  #!/usr/bin/perl

 1  # This is code example 4 in the Graphics-HOWTO
 2  use strict;
 3  use lib "$ENV{HOME}/projects/bioperl-live";
 4  use Bio::Graphics;
 5  use Bio::SearchIO;

 6  my $file = shift or die "Usage: render_blast4.pl <blast file>\n";

 7  my $searchio = Bio::SearchIO->new(-file   => $file,
 8                                    -format => 'blast') or die "parse failed";


 9  my $result = $searchio->next_result() or die "no result";

10  my $panel = Bio::Graphics::Panel->new(-length    => $result->query_length,
11                                        -width     => 800,
12                                        -pad_left  => 10,
13                                        -pad_right => 10,
14                                       );

15  my $full_length = Bio::SeqFeature::Generic->new(-start=>1,-end=>$result->query_length,
16                                                  -display_name=>$result->query_name);
17  $panel->add_track($full_length,
18                    -glyph   => 'arrow',
19                    -tick    => 2,
```

```
20                    -fgcolor => 'black',
21                    -double  => 1,
22                    -label   => 1,
23                   );

24  my $track = $panel->add_track(-glyph       => 'graded_segments',
25                                -label       => 1,
26                                -connector   => 'dashed',
27                                -bgcolor     => 'blue',
28                                -font2color  => 'red',
29                                -sort_order  => 'high_score',
30                                -description => sub {
31                                  my $feature = shift;
32                                  return unless $feature->has_tag('description');
33                                  my ($description) = $feature->each_tag_value('description');
34                                  my $score = $feature->score;
35                                  "$description, score=$score";
36                                 });

37  while( my $hit = $result->next_hit ) {
38    next unless $hit->significance < 1E-20;
39    my $feature = Bio::SeqFeature::Generic->new(-score   => $hit->raw_score,
40                                                -display_name => $hit->name,
41                                                -tag     => {
42                                                             description => $hit->description
43                                                            },
44                                               );
45    while( my $hsp = $hit->next_hsp ) {
46      $feature->add_sub_SeqFeature($hsp,'EXPAND');
47    }

48    $track->add_feature($feature);
49  }

50  print $panel->png;
```

In lines 6-8 we read the name of the file that contains the BLAST results from the command line, and pass it to `Bio::SearchIO->new()`, returning a `Bio::SearchIO` object. We read a single result from the searchIO object (line 9). This assumes that the BLAST output file contains a single run of BLAST only.

We then initialize the panel and tracks as before. The only change here is in lines 24-36, where we create the track for the BLAST hits. The *`-description`* option has now been enhanced to create a line of text that incorporates the "description" tag from the feature object as well as its similarity score. There's also a slight change in line 26, where we introduce the *`-connector`* option. This allows us to configure a line that connects the segments of a discontinuous feature, such as the HSPs in a BLAST hit. In this case, we asked the rendering engine to produce a dashed connector line.

The remainder of the script retrieves each of the hits from the BLAST file, creates a Feature object representing the hit, and then retrieves each HSP and incorporates it into the feature. Line 37 begins a `while()` loop that retrieves each of the similarity hits in turn. We filter the hit by its significance, throwing out any that have an expectation value greater than 1E-20 (you will have to adjust this in your own utilities). We then use the information in the hit to construct a `Bio::SeqFeature::Generic` object (lines 39-44). Notice how the name of the hit and the score are used to initialize the feature, and how the description is turned into a tag named "description."

The start and end bounds of the hit are determined by the union of its HSPs. We loop through each of the hit's HSPs by calling its `next_hsp()` method, and add each HSP to the newly-created hit feature by calling the feature's `add_sub_SeqFeature()` method (line 46). The *`EXPAND`* parameter instructs the feature to expand its start and end coordinates to enclose the added subfeature.

Once all the HSPs are added to the feature, we insert the feature into the track as before using the track's `add_feature()` function.

Figure 4 shows the output from a sample BLAST hit file.

Figure 5. Output from the BLAST parsing and rendering script

The next section will demonstrate how to parse and display feature tables from GenBank and EMBL.

IMPORTANT NOTE: Remember that if you are on a Windows platform, you need to put STDOUT into binary mode so that the PNG file does not go through Window's carriage return/linefeed transformations. Before the final print statement, write binmode(STDOUT).

7. Rendering Features from a GenBank or EMBL File

With `Bio::Graphics` you can render the feature table of a GenBank or EMBL file quite easily. The trick is to use `Bio::SeqIO` to generate a set of `Bio::SeqFeatureI` objects, and to use those features to populate tracks. For simplicity's sake, we will sort each feature by its primary tag (such as "exon") and create a new track for each tag type.

Code example 5 shows the code for rendering an EMBL or GenBank entry.

Example 5. The embl2picture.pl script turns any EMBL or GenBank entry into a graphical rendering

```
 0  #!/usr/bin/perl

 1  # file: embl2picture.pl
 2  # This is code example 5 in the Graphics-HOWTO
 3  # Author: Lincoln Stein

 4  use strict;
 5  use lib "$ENV{HOME}/projects/bioperl-live";
 6  use Bio::Graphics;
 7  use Bio::SeqIO;
 8  use Bio::SeqFeature::Generic;

 9  my $file = shift                       or die "provide a sequence file as the argument";
10  my $io = Bio::SeqIO->new(-file=>$file) or die "couldn't create Bio::SeqIO";
11  my $seq = $io->next_seq                or die "couldn't find a sequence in the file";
12  my $wholeseq = Bio::SeqFeature::Generic->new(-start=>1,-end=>$seq->length,
13                                               -display_name=>$seq->display_name);

14  my @features = $seq->all_SeqFeatures;

15  # partition features by their primary tags
16  my %sorted_features;
17  for my $f (@features) {
18    my $tag = $f->primary_tag;
19    push @{$sorted_features{$tag}},$f;
20  }

21  my $panel = Bio::Graphics::Panel->new(
22                                        -length    => $seq->length,
23                                        -key_style => 'between',
24                                        -width     => 800,
25                                        -pad_left  => 10,
26                                        -pad_right => 10,
27                                        );
28  $panel->add_track($wholeseq,
29                    -glyph => 'arrow',
30                    -bump => 0,
31                    -double=>1,
32                    -tick => 2);

33  $panel->add_track($wholeseq,
34                    -glyph  => 'generic',
35                    -bgcolor => 'blue',
36                    -label  => 1,
37                   );

38  # general case
39  my @colors = qw(cyan orange blue purple green chartreuse magenta yellow aqua);
40  my $idx    = 0;
41  for my $tag (sort keys %sorted_features) {
```

```
42    my $features = $sorted_features{$tag};
43    $panel->add_track($features,
44                      -glyph    =>  'generic',
45                      -bgcolor  =>  $colors[$idx++ % @colors],
46                      -fgcolor  => 'black',
47                      -font2color => 'red',
48                      -key      => "${tag}s",
49                      -bump     => +1,
50                      -height   => 8,
51                      -label    => 1,
52                      -description => 1,
53                     );
54  }

55  print $panel->png;
56  exit 0;
```

The way this script works is simple. After the library load preamble, the script reads the name of the GenBank or EMBL file from the command line (line 8). It passes the filename to `Bio::SeqIO`'s `new()` method, and reads the first sequence object from it (lines 9-11). If anything goes wrong, the script dies with an error message.

The returned object is a Bio::SeqI object, which has a length but no defined start or end coordinates. We would like to create a drawable Bio::SeqFeatureI object to use for the scale, so we generate a new Bio::SeqFeature::Generic object that goes from a start of 1 to the length of the sequence. (lines 12-13).

The script reads the features from the sequence object by calling `all_SeqFeatures()`, and then sorts each feature by its primary tag into a hash of array references named `%sorted_features` (lines 14-20).

Next, we create the `Bio::Graphics::Panel` object (lines 21-27). As in previous examples, we specify the width of the image, as well as some extra white space to pad out the left and right borders.

We now add two tracks, one for the scale (lines 28-32) and the other for the sequence as a whole (33-37). As in the earlier examples, we pass `add_track()` the sequence object as the first argument before the options so that the object is incorporated into the track immediately.

We are now ready to create a track for each feature type. In order to distinguish the tracks by color, we initialize an array of 9 color names and simply cycle through them (lines 39-54). For each feature tag, we retrieve the corresponding list of features from `%sorted_features` (line 42) and create a track for it using the "generic" glyph and the next color in the list (lines 43-53). We set the `-label` and `-description` options to the value "1". This signals `Bio::Graphics` that it should do the best it can to choose useful label and description values on its own.

After adding all the feature types, we call the panel's `png()` method to generate a graphic file, which we print to STDOUT. If we are on a Windows platform, we would have to include `binmode(STDOUT)` prior to this statement in order to avoid Windows textmode carriage return/linefeed transformations.

Figure 5 shows an example of the output of this script.

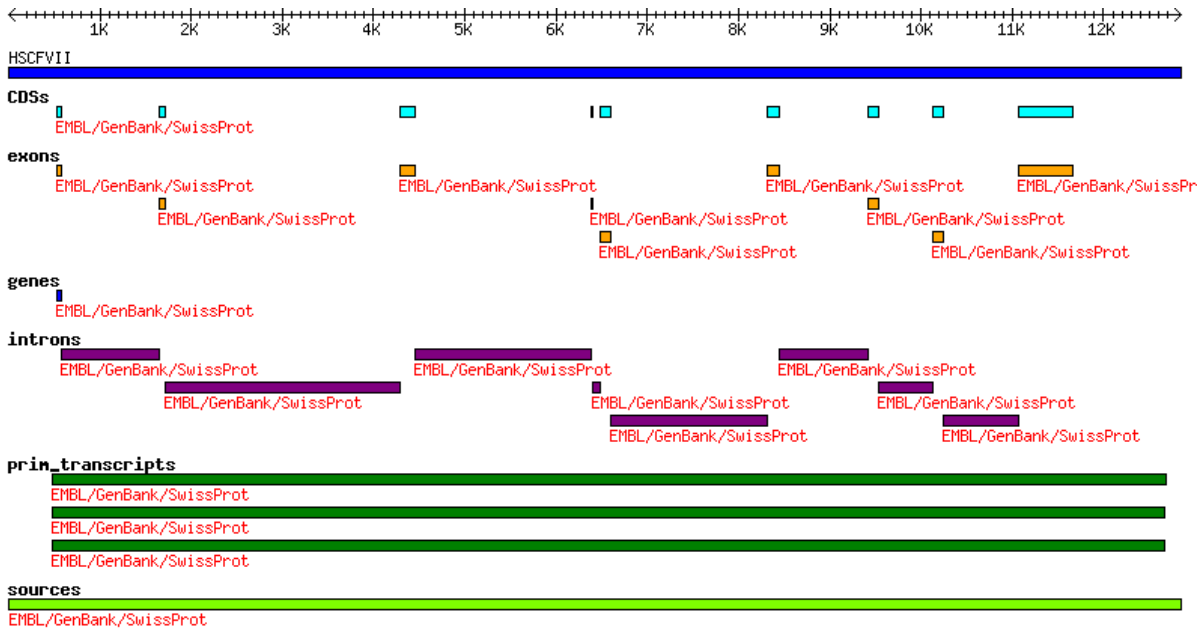

Figure 6. The embl2picture.pl script

8. A Better Version of the Feature Renderer

The previous example's rendering has numerous deficiencies. For one thing, there are no lines connecting the various CDS rectangles in the CDS track to show how they are organized into a spliced transcript. For another, the repetition of the source tag "EMBL/GenBank/SwissProt" is not particularly illuminating.

However, it's quite easy to customize the display, making the script into a generally useful utility. The revised code is shown in example 6.

Example 6. The embl2picture.pl script turns any EMBL or GenBank entry into a graphical rendering

```
 0  #!/usr/bin/perl

 1  # file: embl2picture.pl
 2  # This is code example 6 in the Graphics-HOWTO
 3  # Author: Lincoln Stein

 4  use strict;
 5  use lib "$ENV{HOME}/projects/bioperl-live";
 6  use Bio::Graphics;
 7  use Bio::SeqIO;

 8  use constant USAGE =><<END;
 9  Usage: $0 <file>
10     Render a GenBank/EMBL entry into drawable form.
11     Return as a GIF or PNG image on standard output.

12     File must be in embl, genbank, or another SeqIO-
13     recognized format.  Only the first entry will be
14     rendered.

15  Example to try:
```

```
16     embl2picture.pl factor7.embl | display -

17   END

18   my $file = shift                         or die USAGE;
19   my $io = Bio::SeqIO->new(-file=>$file) or die USAGE;
20   my $seq = $io->next_seq                  or die USAGE;
21   my $wholeseq = Bio::SeqFeature::Generic->new(-start=>1,-end=>$seq->length,
22                                                -display_name=>$seq->display_name);

23   my @features = $seq->all_SeqFeatures;

24   # sort features by their primary tags
25   my %sorted_features;
26   for my $f (@features) {
27     my $tag = $f->primary_tag;
28     push @{$sorted_features{$tag}},$f;
29   }

30   my $panel = Bio::Graphics::Panel->new(
31                                         -length    => $seq->length,
32                                         -key_style => 'between',
33                                         -width     => 800,
34                                         -pad_left  => 10,
35                                         -pad_right => 10,
36                                         );
37   $panel->add_track($wholeseq,
38                     -glyph => 'arrow',
39                     -bump => 0,
40                     -double=>1,
41                     -tick => 2);

42   $panel->add_track($wholeseq,
43                     -glyph  => 'generic',
44                     -bgcolor => 'blue',
45                     -label  => 1,
46                    );

47   # special cases
48   if ($sorted_features{CDS}) {
49     $panel->add_track($sorted_features{CDS},
50                       -glyph      => 'transcript2',
51                       -bgcolor    => 'orange',
52                       -fgcolor    => 'black',
53                       -font2color => 'red',
54                       -key        => 'CDS',
55                       -bump       =>  +1,
56                       -height     =>  12,
57                       -label      => \&gene_label,
58                       -description=> \&gene_description,
59                      );
60     delete $sorted_features{'CDS'};
61   }

62   if ($sorted_features{tRNA}) {
63     $panel->add_track($sorted_features{tRNA},
64                       -glyph     =>  'transcript2',
65                       -bgcolor   =>  'red',
66                       -fgcolor   =>  'black',
67                       -font2color => 'red',
68                       -key       => 'tRNAs',
69                       -bump      =>  +1,
70                       -height    =>  12,
71                       -label     => \&gene_label,
72                      );
73     delete $sorted_features{tRNA};
74   }

75   # general case
76   my @colors = qw(cyan orange blue purple green chartreuse magenta yellow aqua);
77   my $idx    = 0;
78   for my $tag (sort keys %sorted_features) {
79     my $features = $sorted_features{$tag};
80     $panel->add_track($features,
81                       -glyph    =>  'generic',
82                       -bgcolor  =>  $colors[$idx++ % @colors],
83                       -fgcolor  => 'black',
```

```
 84                      -font2color => 'red',
 85                      -key      => "${tag}s",
 86                      -bump     => +1,
 87                      -height   => 8,
 88                      -description => \&generic_description
 89                     );
 90  }

 91  print $panel->png;
 92  exit 0;

 93  sub gene_label {
 94    my $feature = shift;
 95    my @notes;
 96    foreach (qw(product gene)) {
 97      next unless $feature->has_tag($_);
 98      @notes = $feature->each_tag_value($_);
 99      last;
100    }
101    $notes[0];
102  }

103  sub gene_description {
104    my $feature = shift;
105    my @notes;
106    foreach (qw(note)) {
107      next unless $feature->has_tag($_);
108      @notes = $feature->each_tag_value($_);
109      last;
110    }
111    return unless @notes;
112    substr($notes[0],30) = '...' if length $notes[0] > 30;
113    $notes[0];
114  }

115  sub generic_description {
116    my $feature = shift;
117    my $description;
118    foreach ($feature->all_tags) {
119      my @values = $feature->each_tag_value($_);
120      $description .= $_ eq 'note' ? "@values" : "$_=@values; ";
121    }
122    $description =~ s/; $//; # get rid of last
123    $description;
124  }
```

At 124 lines, this is the longest example in this HOWTO, but the changes are straightforward. The major difference occurs in lines 47-61 and 62-74, where we handle two special cases: "CDS" records and "tRNAs". For these two feature types we would like to draw the features like genes using the "transcript2" glyph. This glyph draws inverted V's for introns, if there are any, and will turn the last (or only) exon into an arrow to indicate the direction of transcription.

First we look to see whether there are any features with the primary tag of "CDS" (lines 47-61). If so, we create a track for them using the desired glyph. Line 49 shows how to add several features to a track at creation time. If the first argument to `add_track()` is an array reference, all the features contained in the array will be incorporated into the track. We provide custom code references for the *`-label`* and *`-description`* options. As we shall see later, the subroutines these code references point to are responsible for extracting names and descriptions for the coding regions. After we handle this special case, we remove the CDS feature type from the `%sorted_features` array.

We do the same thing for tRNA features, but with a different color scheme (lines 62-74).

Having dealt with the special cases, we render the remaining feature types using the same code we used earlier. The only change is that instead of allowing `Bio::Graphics::Panel` to guess at the description from the feature's source tag, we use the `-description` option to point to a subroutine that will generate more informative description strings.

The `gene_label()` (lines 93-102) and `gene_description()` (lines 103-114) subroutines are simple. The first one searches the feature for the tags "product" and/or "gene" and uses the first one it finds as the label for the feature. The `gene_description()` subroutine is similar, except that it returns the value of the first tag named "note". If the description is over 30 characters long, it is truncated.

The `generic_description()` (lines 115-124) is invoked to generate descriptions of all non-gene features. We simply concatenate together the names and values of tags. For example the entry:

```
   source          1..12850
                   /db_xref="taxon:9606"
                   /organism="Homo sapiens"
                   /map="13q34"
```

will be turned into the description string "db_xref=taxon:9606; organism=Homo Sapiens; map=13q34".

After adding all the feature types, we call the panel's `png()` method to generate a graphic file, which we print to STDOUT.

Figure 6 shows an example of the output of this script.

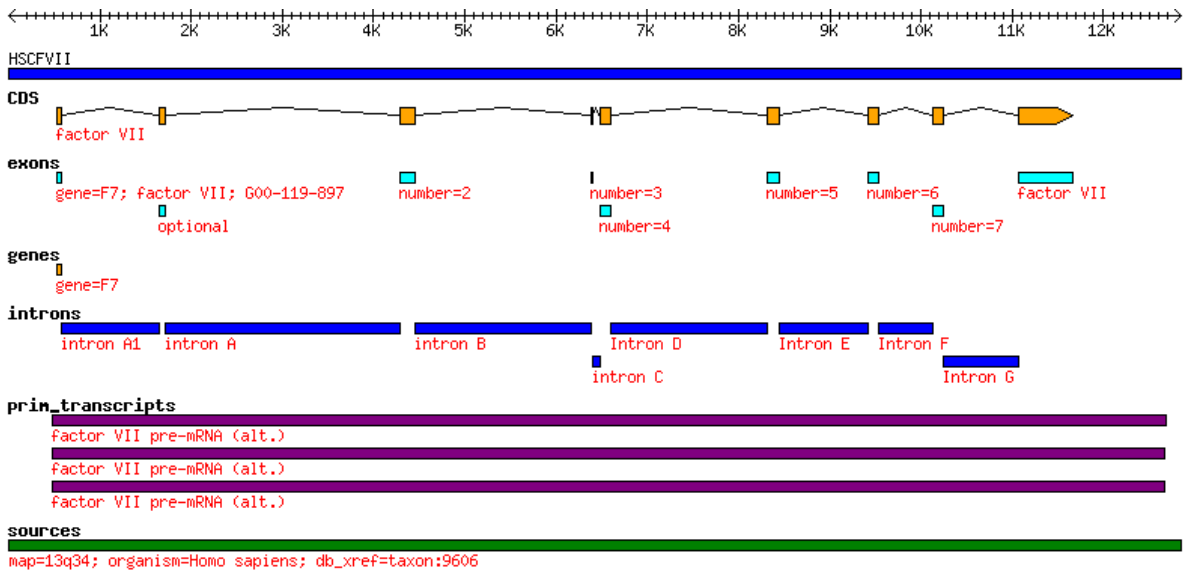

Figure 7. The embl2picture.pl script

9. Summary

In summary, we have seen how to use the `Bio::Graphics` module to generate representations of sequence features as horizontal maps. We applied these techniques to two common problems: rendering the output of a BLAST run, and rendering the feature table of a GenBank/EMBL entry.

The graphics module is quite flexible. In addition to the options that we have seen, there are glyphs for generating point-like features such as SNPs, specialized glyphs that draw GC content and open reading frames, and glyphs that generate histograms, bar charts and other types of graphs. `Bio::Graphics` has been used to represent physical (clone) maps, radiation hybrid maps, EST clusters, cytogenetic maps, restriction maps, and much more.

Although we haven't shown it, `Bio::Graphics` provides support for generating HTML image maps. The Generic Genome Browser [http://www.gmod.org] uses this facility to generate clickable, browsable images of

the genome from a variety of genome databases.

Another application you should investigate is the render_sequence.pl script. This script uses the BioFetch interface to fetch GenBank/EMBL/SwissProt entries dynamically from the web before rendering them into PNG images.

Finally, if you find yourself constantly tweaking the graphic options, you might be interested in `Bio::Graphics::FeatureFile`, a utility module for interpreting and rendering a simple tab-delimited format for sequence features. feature_draw.PLS is a Perl script built on top of this module, which you can find in the scripts/graphics directory in the Bioperl distribution.