bio-samtools Basic Tutorial

Introduction

bio-samtools is a Ruby binding to the popular SAMtools library, and provides access to individual read alignments as well as BAM files, reference sequence and pileup information.

Installation

Installation of bio-samtools is very straightforward, and is accomplished with the Ruby gems command. All you need is an internet connection.

Prerequisites

bio-samtools relies on the following other rubygems:

- FFI
- bio >= 1.4.1

Once these are installed, bio-samtools can be installed with

```
sudo gem install bio-samtools
```

It should then be easy to test whether installation went well. Start interactive Ruby (IRB) in the terminal, and type `require 'bio-samtools'` if the terminal returns `true` then all is well.

```
$ irb
>> require 'bio-samtools'
=> true
```

Working with BAM files

Creating a new SAM object

A SAM object represents the alignments in the BAM file, and is very straightforward to create, you will need a sorted BAM file, to access the alignments and a reference sequence in FASTA format to use the reference sequence. The object can be created and opened as follows:

```
bam = Bio::DB::Sam.new(:bam=>"my_sorted.bam", :fasta=>'ref.fasta')
bam.open
```

Opening the file needs only to be done once for multiple operations on it, access to the alignments is random so you don't need to loop over the entries in the file.

Getting Reference Sequence

Retrieving the reference can only be done if the reference has been loaded, which isn't done automatically in order to save memory. Reference need only be loaded once, and is accessed using reference name, start, end in 1-based co-ordinates. A standard Ruby String object is returned.

```
bam.load_reference
sequence_fragment = bam.fetch_reference("Chr1", 1, 500)
```

Getting Alignments

Alignments can be obtained one at a time by looping over a specified region using the `fetch()` function.

```
bam.load_reference
                        bam.fetch("1",3000,4000).each do |alignment|
                                #do something with the alignment...
                        end
```

A separate method `fetch_with_function()` allows you to pass a block (or a Proc object) to the function for efficient calculation. This example shows a naive conversion of the alignment object to a GFF3 object, which is stored in an array `gff_list`

```
gff_list = []
fetchAlignment = Proc.new do |a|
#what strand is this alignment on...
         a.query_strand ? strand = '+'  : strand = '-'
         gff_list << Bio::DB::GFF3.new(
           :seqid => "Chr1",
           :start => a.pos - 1,
           :end => a.calend,
           :strand => strand,
           :sequence => a.seq,
           :quality => a.qual,
           :feature => 'read',
           :source => 'BWA',
           :phase => '.',
           :score => '.'
         )
         0
       end
       bam.fetch_with_function("Chr1", 3000, 4000, fetchAlignment) #now run the fetch
```

Alignment Objects

The individual alignments represent a single read and are returned as Bio::DB::Alignment objects. These have numerous methods of their own, using `require 'pp'` will allow you to check the attributes contained in each object. Here is an example alignment object. Remember `@` represents a Ruby instance variable and can be accessed as any other method. Thus the `@is_mapped` attribute of an object `a` is accessed `a.is_mapped`

```
require 'pp'
pp an_alignment_object ##some Bio::DB::Alignment object
#<Bio::DB::Alignment:0x101113f80
@al=#<Bio::DB::SAM::Tools::Bam1T:0x101116a50>,
@calend=4067,
@cigar="76M",
@failed_quality=false,
@first_in_pair=false,
@flag=163,
@is_duplicate=false,
@is_mapped=true,
@is_paired=true,
@isize=180,
@mapq=60,
@mate_strand=false,
@mate_unmapped=false,
@mpos=4096,
@mrnm="=",
@pos=3992,
@primary=true,
@qlen=76,
@qname="HWI-EAS396_0001:7:115:17904:15958#0",
@qual="IIIIIIIIIIIIHHIHGIHIDGGGG...",
@query_strand=true,
@query_unmapped=false,
@rname="1",
@second_in_pair=true,
@seq="ACAGTCCAGTCAAAGTACAAATCGAG...",
@tags=
        {"MD"=>#<Bio::DB::Tag:0x101114ed0 @tag="MD", @type="Z", @value="76">,
         "XO"=>#<Bio::DB::Tag:0x1011155d8 @tag="XO", @type="i", @value="0">,
         "AM"=>#<Bio::DB::Tag:0x101116280 @tag="AM", @type="i", @value="37">,
         "X0"=>#<Bio::DB::Tag:0x101115fb0 @tag="X0", @type="i", @value="1">,
         "X1"=>#<Bio::DB::Tag:0x101115c68 @tag="X1", @type="i", @value="0">,
         "XG"=>#<Bio::DB::Tag:0x101115240 @tag="XG", @type="i", @value="0">,
         "SM"=>#<Bio::DB::Tag:0x1011162f8 @tag="SM", @type="i", @value="37">,
         "XT"=>#<Bio::DB::Tag:0x1011162a8 @tag="XT", @type="A", @value="U">,
         "NM"=>#<Bio::DB::Tag:0x101116348 @tag="NM", @type="i", @value="0">,
         "XM"=>#<Bio::DB::Tag:0x101115948 @tag="XM", @type="i", @value="0">}>
```

Getting Coverage Information

Per Base Coverage

It is easy to get the total depth of reads at a given position, the `chromosome_coverage` function is used. This differs from the previous functions in that a start position and length (rather than end position) are passed to the function. An array of coverages is returned, the first position in the array gives the depth of coverage at the given start position in the genome, the last position in the array gives the depth of coverage at the given start position plus the length given

```
coverages = bam.chromosome_coverage("Chr1", 3000, 1000)  #=> [16,16,25,25...]
```

Average Coverage In A Region

Similarly, average (arithmetic mean) of coverage can be retrieved, also with start and length parameters

```
coverages = bam.average_coverage("Chr1", 3000, 1000)  #=> 20.287
```

Getting Pileup Information

Pileup format represents the coverage of reads over a single base in the reference. Getting a Pileup over a region is very easy. Note that this is done with `mpileup` and NOT the now deprecated SAMTools `pileup` function. Calling the `mpileup` method creates an iterator that yields a Pileup object for each base.

```
bam.mpileup do |pileup|
        puts pileup.consensus #gives the consensus base from the reads for that postion
end
```

Pileup options

The `mpileup` function takes a range of parameters to allow SAMTools level filtering of reads and alignments. They are specified as key => value pairs eg

```
bam.mpileup(:r => "Chr1:1000-2000", :Q => 50) do |pileup|
        ##only pileups on Chr1 between positions 1000-2000 are considered,
        ##bases with Quality Score < 50 are excluded
        ...
end
```

Not all the options SAMTools allows you to pass to mpileup are supported, those that cause mpileup to return BCF/VCF are ignored. Specifically these are g,u,e,h,l,L,o,p. The table below lists the SAMTools flags supported and the symbols you can use to call them in the mpileup command.

SAMTools option	description	short symbol	long symbol	default	example
r	limit retrieval to a region	:r	:region	all positions	:r => "Chr1:1000-2000"
6	assume Illumina scaled quality scores	:six	:illumina_quals	false	:six => true
A	count anomalous read pairs scores	:A	:count_anomalous	false	:A => true
B	disable BAQ computation	:B	:no_baq	false	:no_baq => true
C	parameter for adjusting mapQ	:C	:adjust_mapq	0	:C => 25
d	max per-BAM depth to avoid excessive memory usage	:d	:max_per_bam_depth	250	:d => 123
E	extended BAQ for higher sensitivity but lower specificity	:E	:extended_baq	false	:E => true
G	exclude read groups listed in FILE	:G	:exclude_reads_file	false	:G =>

l	list of positions (chr pos) or regions (BED)	:l	:list_of_positions	false	'my_file.txt' :l => 'my_posns.bed'
M	cap mapping quality at value	:M	:mapping_quality_cap	60	:M => 40
R	ignore RG tags	:R	:ignore_rg	false	:R => true
q	skip alignments with mapping quality smaller than value	:q	:min_mapping_quality	0	:q => 30
Q	skip bases with base quality smaller than value	:Q	:imin_base_quality	13	:Q => 30