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:

- bio >= 1.4.2
- bio-svgenes >= 0.4.1

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. BAM files (and hence SAM objects here) are what most of SAMtools methods operate on and are 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')
```

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

The reference is accessed using reference name, start, end in 1-based co-ordinates. A standard Ruby String object is returned.

```
sequence_fragment = bam.fetch_reference("Chr1", 1, 100)
```

The output from this would be the raw sequence as a string, e.g.

```
cctaaccctaaccctaaccctaaccctaaccctaaccctaaccctaaccctaaccctaaccctaaccctaaccctaaccctaaccctaaccctaaccctaacccta
```

A reference sequence can be returned as a Bio::Sequence::NA object buy the use of :as_bio => true

```
sequence_fragment = bam.fetch_reference("Chr1", 1, 100, :as_bio => true)
```

The output from this would be a Bio::Sequence::NA object, which provides a fasta-formatted string when printed

```
>Chr1:1-100
cctaaccctaaccctaaccctaaccctaaccctaaccctaaccctaaccctaaccctaaccctaaccctaaccctaaccctaaccctaaccctaaccctaacccta
```

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 Alignments

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

```
bam.fetch("Chr1",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 takes an alignment object and returns an array of sequences which exactly match the reference.

```
#an array to hold the matching sequences
exact_matches = []

matches = Proc.new do |a|
    #get the length of each read
    len = a.seq.length
    #get the cigar string
    cigar = a.cigar
    #create a cigar string which represents a full-length match
    cstr = len.to_s << "M"
    if cigar == cstr
        #add the current sequence to the array if it qualifies
        exact_matches << a.seq
    end
end

bam.fetch_with_function("Chr1", 100, 500, &matches)

puts exact_matches
```

Alignment stats

The SAMtools flagstat method is implemented in bio-SAMtools to quickly examine the number of reads mapped to the reference. This includes the number of paired and singleton reads mapped and also the number of paired-reads that map to different chromosomes/contigs.

```
bam.flag_stats()
```

An example output would be

```
34672 + 0 in total (QC-passed reads + QC-failed reads)
0 + 0 duplicates
33196 + 0 mapped (95.74%:nan%)
34672 + 0 paired in sequencing
17335 + 0 read1
17337 + 0 read2
31392 + 0 properly paired (90.54%:nan%)
31728 + 0 with itself and mate mapped
1468 + 0 singletons (4.23%:nan%)
0 + 0 with mate mapped to a different chr
0 + 0 with mate mapped to a different chr (mapQ>=5)
```

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 with the `average_coverage` method.

```
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
```

Caching pileups

A pileup can be cached, so if you want to execute several operations on the same set of regions, mpilup won't be executed several times. Whenever you finish using a region, call mpileup_clear_cache to free the cache. The argument 'Region' is required, as it will be the key for the underlying hash. We asume that the options (other than the region) are constant. If they are not, the cache mechanism may not be consistent.

```
#create an mpileup
reg = Bio::DB::Fasta::Region.new
reg.entry = "Chr1"
reg.start = 1
reg.end = 334

bam.mpileup_cached(:r=>reg,:g => false, :min_cov => 1, :min_per =>0.2) do |pileup|
    puts pileup.consensus
end
bam.mpileup_clear_cache(reg)
```

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 will return a Pileup object, those that cause mpileup to return BCF/VCF will be ignored. Specifically these are g,u,e,h,I,L,o,p. The table below lists the SAMtools flags supported and the symbols you can use to call them in the mpileup command.

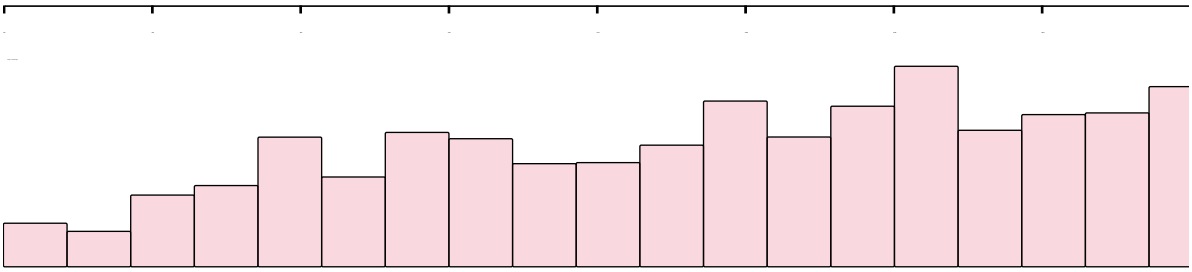
SAMtools options	description	short symbol	long symbol	default	example
r	limit retrieval to a region	:r	:region	all positions	:r => "Chr1:1000-2000"
6	assume Illumina scaled quality scores	:six	:illumina_quals	false	:six => true
A	count anomalous read pairs scores	:A	:count_anomalous	false	:A => true
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 => my_file.txt
l	list of positions (chr pos) or regions (BED)	:l	:list_of_positions	false	: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

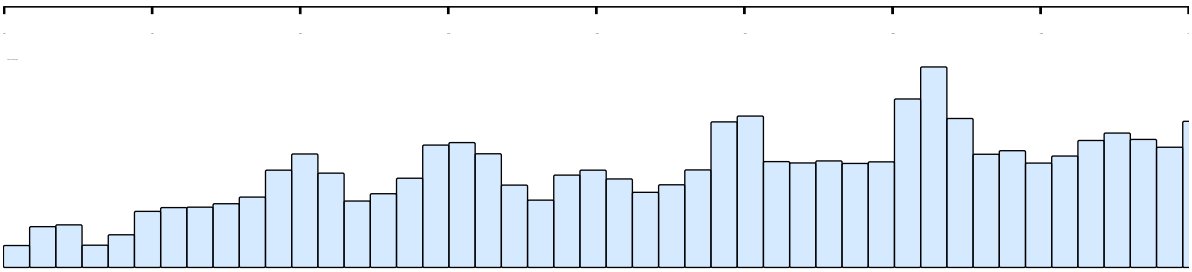
Coverage Plots

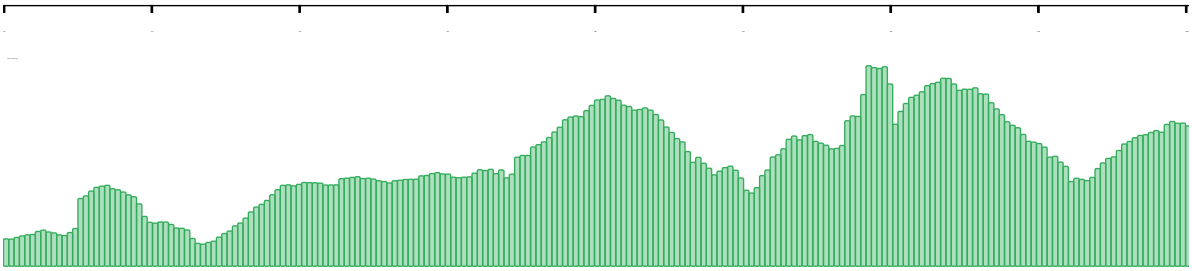
You can create images that represent read coverage over binned regions of the reference sequence. The output format is svg. A number of parameters can be changed to alter the style of the plot. In the examples below the bin size and fill_color have been used to create plots with different colours and bar widths.

The following lines of code...

```
bam.plot_coverage("Chr1", 201, 2000, :bin=>20, :svg => "out2.svg", :fill_color => '#F1A1B1')
bam.plot_coverage("Chr1", 201, 2000, :bin=>50, :svg => "out.svg", :fill_color => '#99CCFF')
bam.plot_coverage("Chr1", 201, 1000, :bin=>250, :svg => "out3.svg", :fill_color => '#33AD5C', :stroke => '#33AD5C')
```







VCF methods

For enhanced snp calling, we've included a VCF class which reflects each non-metadata line of a VCF file. The VCF class returns the eight fixed fields present in VCF files, namely chromosome, position, ID, reference base, alt bases, alt quality score, filter and info along with the genotype fields, format and samples. This information allows the comparison of variants and their genotypes across any number of samples. The following code takes a number of VCF objects and examines them for homozygous alt (1/1) SNPs

```
vcfs = []
vcfs << vcf1 = Bio::DB::Vcf.new("20 14370   rs6054257   G   A   29  0   NS=3;DP=14;AF=0.5;DB;H2 GT:GQ:DP:HQ 0|0:48:1:51,51
1|0:48:8:51,51  1/1:43:5:-1,-1") #from a 3.3 vcf file
vcfs << vcf2 = Bio::DB::Vcf.new("19 111 .   A   C   9.6 .   .   GT:HQ   0|0:10,10   0/0:10,10   0/1:3,3") #from a 4.0 vcf file
vcfs << vcf3 = Bio::DB::Vcf.new("20 14380   rs6054257   G   A   29  PASS    NS=3;DP=14;AF=0.5;DB;H2 GT:GQ:DP:HQ 0|0:48:1:51,51
1|0:48:8:51,51  1/1:43:5:.,") #from a 4.0 vcf file

vcfs.each do |vcf|
    vcf.samples.each do |sample|
        genotype = "#{sample[1]['GT']}"
        if genotype == '1/1' or genotype == '1|1'
            print vcf.chrom, " "
            puts vcf.pos
        end
    end
end

=> 20 14370
=> 20 14380
```

Tests

The easiest way to run the built-in unit tests is to change to the bio-SAMtools source directory and running 'rake test'

Each test file tests different aspects of the code.