Version 1.1.0 Last updated 29th October 2014

BamM is a c library, wrapped in python, that parses BAM files.

The code is intended to provide a faster, more stable interface to parsing BAM files than PySam, but doesn't implement all / any of PySam's features.

Do you want all the links that join two contigs in a BAM?

Do you need to get coverage?

Would you like to just work out the insert size and orientation of some mapped reads?

Then BamM is for you!

For impatient people

1. `$ BamM make* <reference.fna> -c read1.R1.fq.gz read1.R2.fq.gz ...`
2. `$ BamM parse -c covs.tsv -l links.tsv -i inserts.tsv mapping.bam`
3. `$ BamM extract -g BIN_1.fna -b mapping.bam`

*BamM make produces indexed and sorted BAM files which contain only reads that mapped.

Table of Contents

Installation

The BAM parsing is done using c and a few external libraries. This slightly complicates the BamM installation process but fortunately not too much.

Dependencies

If you're running 'BamM make' you'll need to have BWA and Samtools installed. Installation of these tools is really straightforward. You can find the code and instructions at:

Samtools:	https://github.com/samtools/samtools
BWA:	https://github.com/lh3/bwa

If you're installing system-wide then you can use your favourite package manager to install htslib and libcfu. For local installs, or installs that will work with the Linux "modules" system, you need to be a bit trickier. This is the type of install I'll be documenting here.

The notes here are for installing on Ubuntu, but should be transferable to any other Linux system. I'm not sure if these notes are transferable to fashionable overpriced systems with rounded rectangles and retina displays and I'm almost certain you'll need some sysadmin-fu to get things going on a Windows system. If you do get it all set up then please let me know how and I'll buy you a 5 shot venti, 2/5th decaf, ristretto shot, 1 pump Vanilla, 1 pump Hazelnut, breve, 1 sugar in the raw, with whip, caramel drizzle on top, free poured, 4 pump mocha.

First, you need git, zlib and a C-compiler. On Ubuntu this looks like:

```
$ sudo apt-get -y install git build-essential zlib1g-dev
```

Next you'll need htslib (Samtools guts) and libcfu (hash objects) (if you haven't already installed them system wide)

Installing htslib

Get the latest htslib from github:

```
$ git clone https://github.com/samtools/htslib.git
```

For various reasons we need to install a statically linked version of htslib. When making use this command instead of just 'make':

```
$ make CFLAGS='-g -Wall -O2 -fPIC -static-libgcc -shared'
```

Installing libcfu

I have built this library around libcfu 0.03. It is available here:

http://downloads.sourceforge.net/project/libcfu/libcfu/libcfu-0.03/libcfu-0.03.tar.gz

On my system I have trouble installing it because of inconsistencies with print statements. I fix this with sed like so, and then install locally:

```
$ tar -xvf libcfu-0.03.tar.gz
$ cd libcfu-0.03
$ sed -i -e "s/%d/%zd/g" examples/*.c
$ sed -i -e "s/%u/%zu/g" examples/*.c
# Remove the '--prefix=' part to install system wide
$ ./configure --prefix=`pwd`/build
# Code in the examples folder breaks compilation, nuke from the Makefile
$ sed -i -e "s/src examples doc/src doc/" Makefile
# make and install
$ make CFLAGS='-g -Wno-unused-variable -O2 -fPIC -static-libgcc -shared'
$ make install
```

If you install these libraries to local folders (e.g. in your home folder) then you need to take note of where you installed them. If you installed them system-wide then it *should* be no hassle.

Installing BamM

Get the latest version from github (pip hates this code for some reason...):

```
$ git clone https://github.com/minillinim/BamM.git
```

If you installed htslib and libcfu system-wide then installation is very straight forward:

```
$ python setup.py install
```

If you installed one or more of these libraries locally then you need to tell setup.py where they are:

```
$ python setup.py install --with-libhts-lib /path/to/htslib
--with-libhts-inc /path/to/htslib --with-libcfu-inc /path/to/libcfu/include/
--with-libcfu-lib path/to/libcfu/lib/
```

Relative paths are OK. You can add the --prefix flag to setup.py to install BamM locally. Once done, don't forget to add BamM to your PYTHONPATH. Also, if htslib and libcfu are in non-standard places and you plan to access the C code, you'll need to mess with your LD_LIBRARY_PATH.

BamM command line overview

I've wrapped the python in a script / library called BamM.

BamM has 3 modes; 'make', 'parse' and 'extract'. The first option allows you to make BAM files. The second option lets you derive coverage profiles or linking information. The final option lets you extract reads that map to a set(s) of contigs.

Making BAM files

BamM make

Description:

Map several read sets onto a single reference with one command.

Required arguments:

-d –database <file.fna>	name of fna file to map reads onto

And at least one of:

-i --interleaved <interleaved.fa> [<interleaved.fa> ...]	shuffled reads
-c --coupled <coupled.fa> [<coupled.fa> …]	paired files
-s --single <single.fa> [<single.fa> ...]	singleton read files

Optional arguments:

-p --prefix	''	prefix to apply to BAM files
-o --out_folder	'.'	write to this folder
--index_algorithm	auto	algorithm bwa uses for indexing
--alignment_algorithm	mem	algorithm bwa uses for alignment
-k --keep	False	keep all the database index files etc after
-K --kept	False	assume the indices already exist, don't re-make
-f --force	False	force overwrite of index files if they are present
--output_tam	False	output TAM file instead of BAM file
-v --verbose	False	be verbose
-t –threads	1	max number of threads to use
-m --memory	2GB/thread	maximum memory to use per bwa process

Example usage:

```
$ BamM make -d my_assembly.fa -i ileaved_1.fastq.gz ileaved_2.fastq.gz -c
paired_R1.fastq.gz paired_R2.fastq.gz -s unpaired.fastq.gz [-t 20] [-v]
```

This command will make 4 BAM files by mapping the two interleaved read sets, the one paired read set and the singleton read set onto the reference my_assembly.fa

The code calls BWA and Samtools to produce a set of sorted and indexed BAM files. If you specify -t <threads> then BamM will pass this onto BWA and Samtools.

Use -v to get more verbose output.

NOTE: To save space, the final BAM files contain only mapped reads.

NOTE: Output files are automatically named based on the names of the read files and the database, however you can specify the output directory and a prefix to append to the beginning of all output files.

Getting information from a collection of BAM files

BamM parse

Description:

BamM parse lets you find the insert size(s) and relative read orientation(s) associated with a collection BAM files, it lets you find paired reads that link contigs together and it also lets you create coverage profiles of individual contigs across multiple BAM files.

Required arguments:

-b –bamfiles <bamfile> [<bamfile> …]	space separated list of BAM files to parse

Optional arguments:

-l --links	''	filename to write pairing links to
-i –inserts	''	filename to write insert distributions to
-c –coverages	''	filename to write coverage profiles to
-n –num_types	1	number of insert/orientation types per BAM
-m --coverage_mode	pmean	how to calculate coverage* (req --coverages)
-r –cutoff_range	?	used to calculate upper / lower rejection cut offs when calculating coverage
--length	ANY	minimum Q length
--base_quality	ANY	base quality threshold (Qscore)
--mapping_quality	ANY	mapping quality threshold
--max_distance	ANY	maximum allowable edit distance from query to reference
--use_secondary	False	use reads marked with the secondary flag
--use_supplementary	False	use reads marked with the supplementary flag
-v, --verbose	False	be verbose
-t –threads	1	maximum number of threads to use

The 'cutoff_range' variable is used for trimmed mean and outlier mean. This argument takes at most two values. The first is the lower cut off and the second is the upper. If only value is supplied then lower == upper.

Example usage:

Finding the insert size and relative orientation of paired reads

```
$ BamM parse file.bam
```

Produces output like this:

```
#file     insert     stdev   orientation   supporting
file.bam  899.7514   14.7167 IN            10000
```

The *IN* orientation indicates that this is an Illumina-style paired-end (PE) library with an insert of ~900 bp and a standard deviation of ~15 bp. Illumina-style mate pair (MP) libraries will typically have orientation *OUT*.

Many MP libraries also have a shadow library which looks like someone added some PE reads to the mix. You can tell BamM to look for more than one insert type by specifying the -n option:

```
$ BamM parse -n 2 mate_pair_file.bam
```

Produces output like this:

```
#file                insert      stdev     orientation  supporting
mate_pair_file.bam   2524.4540   729.1291  OUT          10000
mate_pair_file.bam   251.6253    44.7241   IN           10000
```

Multiple BAM files are separated using spaces. The -n argument is space separated too. By default BamM prints this info to stdout. Use the -i argument to specify a file to write the results to.

```
$ BamM parse pe_file.bam mp_file.bam -n 1 2 -i inserts.tsv
```

This command will analyse the reads in pe_file.bam and try to find one insert type and the reads in mp_file.bam and try to find two insert types. The resulting table will be written to the file inserts.tsv.

Creating a coverage profile from several BAM files

When passed the '-c <filename>' argument, BamM will produce a table of coverage values for each BAM file. This is referred to as a coverage profile.

```
$ BamM parse -c coverage.tsv -m <COV_MODE> f1.bam f2.bam f3.bam
```

Produces this output in the file 'coverage.tsv'

```
#contig     Length   f1.bam       f2.bam       f3.bam
contig_1    946      103.0000     327.0000     369.0000
contig_3    1147     130.0000     492.0000     778.0000
contig_5    1465     228.0000     643.0000     970.0000
contig_7    168      34.0000      82.0000      102.0000
contig_9    4045     899.0000     1756.0000    2649.0000
```

The -t option indicates the maximum number of threads BamM will use. This option speeds up the process but you should only use as may threads as you have BAM files. If you have 6 BAM files then you'll see no improvement when using -t 6, -t 7 or -t 700.

Coverage calculation modes

BamM implements several coverage calculation methods. The user can choose the method using the -m argument.

opmean: Outlier pileup coverage: average of reads overlapping each base, after bases with coverage outside mean +/- 1 standard deviation have been excluded. The number of standard deviation used for the cutoff can be changed with --coverage_range.

pmean: Pileup coverage: average of number of reads overlapping each base

tpmean: Trimmed pileup coverage: average of reads overlapping each base, after bases with in the top and bottom 10% have been excluded. The 10% range can be changed using --coverage_range.

counts: Absolute number of reads mapping

cmean: Like 'counts' except divided by the length of the contig

pmedian: Median pileup coverage: median of number of reads overlapping each base

Finding reads that link 2 contigs

When passed the '-l <filename>' argument, BamM will find paired reads that link contigs.

```
$ BamM parse -l links.tsv f1.bam f2.bam f3.bam
```

Produces this output in the file links.tsv:

```
#cid_1       cid_2        len_1  pos_1  rev_1  len_2  pos_2  rev_2  file
contig_2203  contig_3479  1664   334    0      3873   2866   0      f2.bam
contig_2203  contig_3479  1664   384    0      3873   2818   0      f2.bam
contig_2203  contig_3479  1664   383    0      3873   2831   0      f2.bam
contig_2203  contig_3479  1664   349    0      3873   2864   0      f2.bam
contig_2203  contig_3479  1664   338    0      3873   2862   0      f2.bam
```

The first (non-header) line is interpreted like this:

contig_2203 is linked to contig_3479.

The first read is towards the start of contig_2203 (len_1 == 1664, pos_1 == 334) and is in the same orientation as the contig (rev_1 == 0)

The second (paired) read is towards the end of contig_3479 (len_2 == 3873, pos_2 == 2866) and is also in the same orientation as the contig (rev_2 == 0)

The linking information was extracted from file: f2.bam.

The lines following this one describe other links between the two contigs.

NOTE: the -i, -c and -l options are not mutually exclusive and can be run at the same time.

Extracting reads from BAM files

BamM extract

Description:

Extract reads that map to collections of contigs from a collection of BAM files.

Required arguments:

-g --groups <group> [<group> ...]	files containing reference names (1 per line) or contigs file in fasta format
-b --bamfiles <file> [<file> ...]	BAM files to parse

Optional arguments:

-p --prefix	''	prefix to apply to output files
-o --out_folder	'.'	write to this folder
--mix_bams	False	use the same file for multiple BAM files
--mix_groups	False	use the same files for multiple group groups
--mix_reads	False	use the same files for paired/unpaired reads

--interleave	False	interleave paired reads in ouput files
--mapping_quality	0	mapping quality threshold
--use_secondary	False	use reads marked with the secondary flag
--use_supplementary	False	use reads marked with the supplementary flag
--max_distance	1000	maximum edit distance from query to reference
--no_gzip	False	do not gzip output files
--headers_only	False	extract only (unique) headers
-v, --verbose	False	be verbose
-t --threads	1	maximum number of threads to use

Example usage:

Extract all reads mapping to a particular set of contigs

```
$ BamM extract -g group1.file group2.file -b f1.bam f2.bam f3.bam
```

Will extract all reads from each of the three BAM files that map to the contigs in group1 or group2. The 'group' files can be multiple (gzipped) FASTA (like the fna files you can extract from GroopM) or lists of contig headers (one sequence per line).

Unless specified otherwise, BamM differentiates between paired and unpaired reads (from a mapping and group perspective), reads from different BAM files and reads mapping to contigs in different groups.

Paired reads may not be paired when mapped (only one read maps). Also paired reads may map to different groups / bins, so when they're extracted they are *unpaired* in a *group* sense. BamM preserves this (and other) information in the read header.

The read header has the following format:

>g_<group>**;p_**<pairing_info>**;b_**<bamfile>**;c_**<contig_id>**;r_**<read_id>

Where:

<group>	the name of the group / bin file with the contig this read maps to.
<pairing_info>	describes the pairing information about this read (see below).
<bamfile>	the name of the BAM file containing this read.
<contig_id>	the id of the contig the read maps to.
<read_id>	the id of the read as given in the BAM file.

The pairing information has the following format:

p_<code>R_<code>M_<code>G;

Where <code> is one of:

P	Paired	N	Not applicable
U	Unpaired	E	Error

For example:

p_PR_PM_PG;	indicates a paired read that is mapped as a pair to contig(s) within the same group.
p_PR_PM_UG;	indicated a paired read that is mapped as a pair to contig(s) that are in different groups.
p_PR_UM_NG;	indicates a paired read where only this read was mapped.
p_UR_NM_NG;	indicates an unpaired read.

NOTE: this command can produce A LOT of output files.

Using BamM as a python library

BamM is intended to be used as a python library in any code that needs to produce coverage profiles or paired read linking information. The dev docs will be more useful than this quick guide. But it's here to give a taste of what you can do with BamM.

Calculating coverage profiles, insert types or linking reads

The following snippet shows how to calculate a coverage profile within your code.

```
# first import it
from BamM.bamParser import BamParser
from bamm.bamFile import BM_coverageType

# choose the type of coverage to calculate and make a parser
cov_type = BM_coverageType(CT.P_MEAN_OUTLIER, 1, 1)
BP = BamParser(cov_type)

# get a list of BAM files and parse them
bam_files = ['file1.bam', 'file2.bam']
BP.parseBams(bam_files,
             doLinks=False,      # set to False for no links
             doCovs=True,        # set to False for no coverages
             threads=2)          # 2 files so use 2 threads
```

The BamParser has an instance variable called a BamFileInfo (BFI). This object contains all the information that has been derived from the BAM files.

BP.BFI.numBams	the number of BAMs that were parsed
BP.BFI.bamFiles	path to the BAM files
BP.BFI.numContigs	the number of contigs in the BAM files
BP.BFI.contigLengths	array of contig lengths
BP.BFI.contigNames	array of contig names
BP.BFI.coverages	matrix of coverage values (numContigs x numBams)
BP.BFI.links	hash of links (see dev docs for more information)

The BamParser comes with several pre-written print functions:

BP.printBamTypes()	print insert information as detailed above
BP.printCoverages()	print coverage information
BP.printLinks()	print linking read information

All of these functions print to stdout. If you supply a file name then BamM will print the information there.