REAPR version 1.0.18

Martin Hunt

Feb 23$^{\text{rd}}$ 2015

Contents

1 Installation

1.1 Prerequisites

You will need R [6] to be already installed and in your path, in addition to these Perl modules:

- `File::Basename`
- `File::Copy`
- `File::Spec`
- `File::Spec::Link`
- `Getopt::Long`
- `List::Util`

REAPR uses several free software packages in order to run, most of which come bundled with the REAPR code and do not need to be installed separately. The bundled tools are BamTools [1], SAMtools [4], Tabix [3] (C++ version (`https://github.com/ekg/tabixpp`)) and SNP-o-matic [5]. REAPR can make files for viewing in Artemis [2], therefore we recommend installing Artemis (at least version 15.0.0) for viewing the results.

1.2 Install REAPR

Check that you have R and the Perl modules listed above installed, then download the tarball and run:

```
tar -zxf Reapr_1.0.18.tgz
cd Reapr_1.0.18
./install.sh
```

Then add `reapr` to your `$PATH`. You can check that the installation worked by downloading the test data from `ftp://ftp.sanger.ac.uk/pub4/resources/software/reapr/Reapr_1.0.18.test_data.tar.gz`. Then run the following.

```
tar -zxf Reapr_1.0.18.test_data.tar.gz
cd Reapr_1.0.18.test_data
./test.sh
```

2 Brief instructions

REAPR uses several third-party tools, some of which can fall over on certain contig names. Therefore it is a good idea *before* you do anything else (like mapping reads to your assembly) to run

```
  reapr facheck assembly.fa
```

If it returns no errors, then run the pipeline. Otherwise, rename your contigs with

```
  reapr facheck assembly.fa new_assembly
```

The prerequisites for REAPR are as follows.

1. An assembly in FASTA format `assembly.fa`.

2. A BAM file of paired reads mapped to the assembly. This can be made manually or by using the `smaltmap` function of REAPR, which maps using SMALT with the recommended settings described below. If you have a large number of reads and are using a compute farm to run the mapping yourself in parallel, then you can run `smaltmap` with the `-x` option to see the recommended mapping commands.
   The input BAM file should be sorted by coordinate, indexed and have duplicates either marked or removed. REAPR will ignore reads marked as duplicates. Reads in a pair should be pointing towards each other. We recommend SMALT (`http://www.sanger.ac.uk/resources/software/smalt/`) with the options `-x -r`. This will map reads repetitively (`-r`) and map each reads in a pair independently of each other (`-x`). Independent mapping is important, so that reads in a pair are not incorrectly forced to be mapped near to each other.
   Reads in this BAM file can be from any technology and can have any read length. The only restriction is that they must be paired reads with the correct orientation being to point towards each other.
   The higher your read coverage, the better the results. For minimum coverage, it is best to think in terms of fragment coverage, since REAPR analyses fragments to determine assembly breakpoints. A minimum of ∼15X fragment coverage will suffice.

3. Optionally, high-quality (probably short insert Illumina) paired to use the information when mapping them perfectly and uniquely to the assembly. Doing this will allow REAPR to accurately call error-free bases in the assembly. It will not affect the error calling. The correct orientation of these reads should be to point towards each other.
   By default REAPR will not count a base as error-free if it has less than 5X perfect and unique coverage of these reads. If these high-quality reads were used to do the assembly, then there will almost certainly be enough coverage to use them for REAPR.

If you want to use perfect unique mapping information, run the pipeline with these commands:

```
reapr smaltmap assembly.fa long_1.fq long_2.fq long_mapped.bam
reapr perfectmap assembly.fa short_1.fq short_2.fq i_size perfect_prefix
reapr pipeline assembly.fa long_mapped.bam outdir perfect_prefix
```

where `short_1.fq`/`short_2.fq` are FASTQ files of the short insert read pairs, `long_1.fq`/`long_2.fq` are FASTQ files of the long insert read pairs and `i_size` is the insert size of the short insert reads. Note that the first two commands are independent and can be run in parallel. If you are not using perfect unique mapping information, then run the pipeline with

```
reapr smaltmap assembly.fa long_1.fq long_2.fq long_mapped.bam
reapr pipeline assembly.fa long_mapped.bam outdir
```

REAPR always assumes that the reads are 'innies'. The output files will all be in the directory `outdir`, the most important of which are

1. `03.score.errors.gff.gz`, a report of the errors found;
2. `04.break.broken_assembly.fa`, a new version of the assembly, with scaffolds broken based on the errors found;
3. `05.summary.report.txt`, a summary of the errors found in the assembly, plus contiguity statistics (N50 *etc*) of the original and broken assemblies.

To view a contig of interest, for example `contig1`, with Artemis, use the following two commands.

```
reapr plots -s reapr reapr.stats.gz plot.contig1 assembly.fa contig1
./plot.contig1.run_art.sh
```

The first command generates the necessary files and the second starts Artemis.

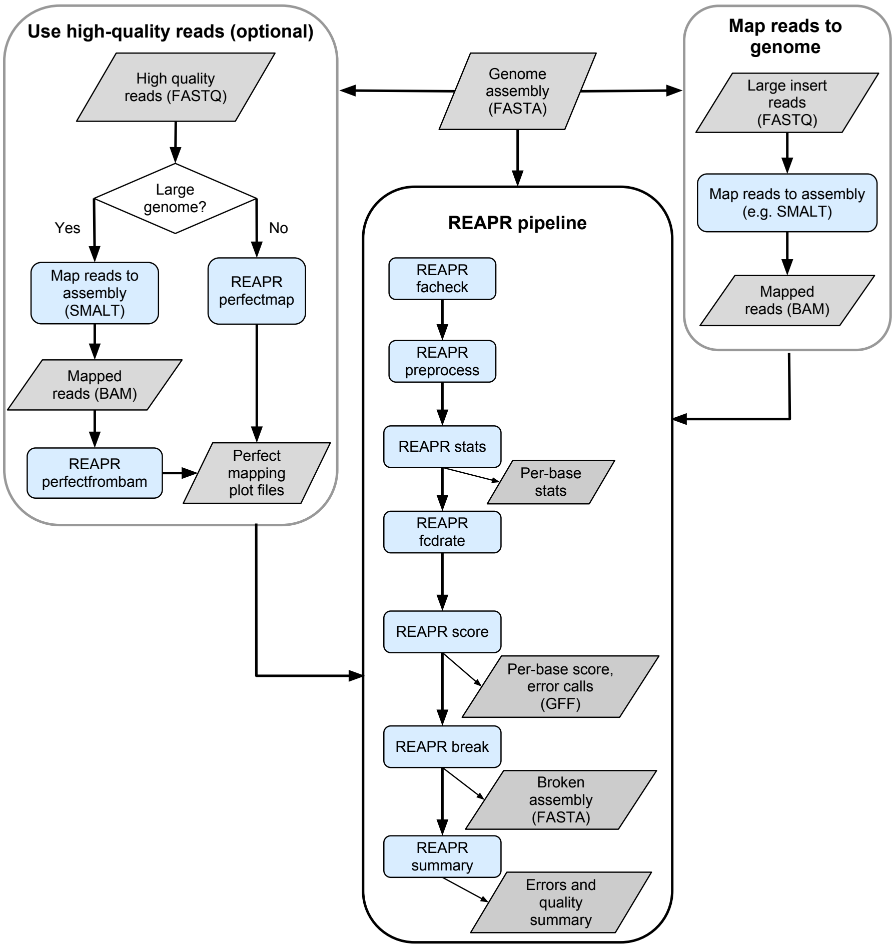

Figure 1: REAPR workflow. Blue background boxes show REAPR tasks (`pipeline` is a shortcut for running `facheck`, `preprocess`, `stats`, `fcdrate`, `score`, `break` and `summary`).

3 Overview

REAPR uses read pairs mapped to an assembly in order to evaluate that assembly. High quality paired reads (typically from a short insert Illumina library) are used to help score each base of the assembly, and large insert data (from any technology) are used to call assembly errors. Use of short insert data is optional. A schematic of the REAPR pipeline is shown in Figure 1.

The error types described later should be self-explanatory, with one exception. REAPR identifies breakpoints in the assembly by analysing fragment coverage and also the fragment coverage distribution (FCD) at each base (see Figure 2). The difference between the expected FCD and actual FCD at each base is referred to as the 'FCD error'. An FCD error usually represents incorrect scaffolding, a large insertion or deletion in the assembly, or sometimes a false join in a contig.

The pipeline needs, as a minimum, a BAM file of paired reads mapped to an assembly

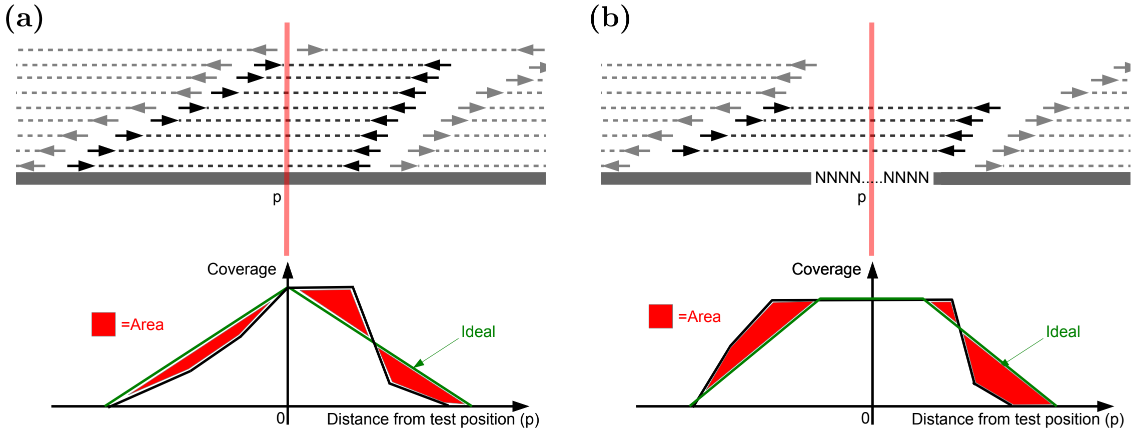
(a)
p
Coverage
=Area
Ideal
0
Distance from test position (p)
(b)
NNNN.....NNNN
p
Coverage
=Area
Ideal
0
Distance from test position (p)

Figure 2: Fragment coverage distribution (FCD), calculated at position p in the genome. Only read pairs, shown in black, spanning p are counted towards the calculation (grey reads are ignored). The resulting fragment coverage is plotted (shown in black) and compared to the ideal shape (green). The difference in area between the two plots is the FCD error, coloured in red. Plots are normalised to have a maximum y value of 1 and the x values are scaled so that the ideal plot has $y = 0$ when $x = \pm 1$. (a) and (b) show the FCD calculation for a base which is not in a gap and in a gap respectively.

to call assembly errors. REAPR will work best if these read pairs are from a large insert library, but any paired data can be used. We recommend that you use the largest insert data available.

Optionally, if you have high-quality reads which are expected to mostly map perfectly to the assembly (typically this would be a short insert Illumina library), these can be used to help score each base of the genome. This is done by mapping the read pairs perfectly and uniquely to the assembly, so that both reads in a pair are mapped only if they match perfectly across their entire length and cannot be mapped to more than one place in the genome.

The general usage is

```
 reapr <task> [options]
```

where `task` is one of `facheck`, `gapresize`, `preprocess`, `perfectmap`, `stats`, `score`, `break`, `plots` or `pipeline`.

REAPR uses several third-party tools, some of which can fall over on certain contig names. Therefore it is a good idea *before* mapping, to run

```
reapr facheck assembly.fa
```

where `assembly.fa` is your FASTA file containing the assembly. If it returns no errors, then it is safe to run the pipeline. Otherwise, rename your contigs with

```
reapr facheck assembly.fa new_assembly
```

Note that the contig names in the assembly FASTA file must exactly match those in the input BAM file. For example, many read mappers will ignore everything after the first whitespace character in a name. `facheck` replaces 'bad' characters with an underscore. If this were to lead to non-unique names in the output, then the offending names will have `.1`, `.2`, ... appended. Hence the recommendation to check your contig names before doing the mapping.

If short insert Illumina reads are available, then first run `perfectmap`. This stage assumes that all reads are of the same length. Then the pipeline can be run (as outlined previously). The stages of the pipeline are `preprocess`, `stats`, `score`, `break` and can be run with one call to `pipeline`.

`preprocess` samples the first million bases of the assembly (excluding gaps), in order to get estimates of various stats that are needed when `stats` is run. The task `stats` needs a BAM file and (if you ran `perfectmap` earlier) a perfect mapping plot file, as input. This will generate statistics at each position in the assembly. After `stats`, the task `score` is run in order to score each position in the genome and make a GFF file of errors in the assembly. This task needs the files made by `stats` in order to run. A broken assembly can be made with `break`, which uses the errors file to make the broken assembly.

A single call to `pipeline` is usually all that is needed. It will work in most cases, however it does not allow any control over the options at each step. If settings other than the defaults are required, it is necessary to run the tasks one by one, changing the parameters at each stage. When running `pipeline`, each system call is printed to `stdout`, so you can see exactly what was run and easily rerun a stage with different options from those chosen by the pipeline.

Finally, the task `plots` can be used to generate Artemis plot files from the statistics. It needs the statistics file made by `stats` to run and can optionally use the scores file made by `scores`. The files made by `plots` can be quite large, and there are several made per contig, so this task is not run by default as part of the pipeline.

4 Examples

How you run REAPR will depend on the input data: whether or not you have short insert reads in order to call error-free bases accurately, and the size of the genome. All the combinations are listed below, with the following filenames:

- `assembly.fa`: FASTA file of the assembly
- `short_1.fq`, `short_2.fq`: FASTQ files of the forward and reverse short insert reads.
- `long_1.fq`, `long_2.fq`: FASTQ files of the forward and reverse large insert reads (could be the same as the short insert reads).

The definition of a 'small' and 'large' genome is probably dependent on memory resources. If you are using short insert reads to generate perfect unique mapping depth, then this is the part that can be high memory. The small genome method is very fast, at the cost of high memory usage and the assumption that all reads are the same length. It used 2GB, 6GB and 26GB of memory on genomes of size 23MB, 100MB and 380MB respectively. The large genome method uses significantly less memory, at the expense of speed.

4.1 Always run facheck

To make sure you have no sequence names that could break the pipeline, run this first:

```
reapr facheck assembly.fa
```

If it returns an error, then make a new FASTA file of your assembly with:

```
reapr facheck assembly.fa assembly_renamed
```

The new FASTA file will be called `assembly_renamed.fa`.

4.2 Small genome, using short and long insert reads

```
reapr perfectmap assembly.fa short_1.fq short_2.fq 300 perfect
reapr smaltmap assembly.fa long_1.fq long_2.fq long_mapped.bam
reapr pipeline assembly.fa long_mapped.bam output_directory perfect
```

where we have short insert reads with an insert size of 300bp.

4.3 Large genome, using short and long insert reads

```
reapr smaltmap assembly.fa short_1.fq short_2.fq short_mapped.bam
reapr perfectfrombam short_mapped.bam perfect 100 500 3 4 76
reapr smaltmap assembly.fa long_1.fq long_2.fq long_mapped.bam
reapr pipeline assembly.fa long_mapped.bam output_directory perfect
```

In this example, reads in a pair pointing towards each other, in the insert size range 100–500bp, with mapping quality at least 4 and alignment score at least 76 would be used to generate the perfect and uniquely mapped depth at each base of the genome. Reads will be counted as repetitive if their mapping quality is $\leqslant 3$. Note that mapping quality scores are mapper-dependent; this example has suitable values for SMALT. Most mappers give a low mapping quality to a read which maps to multiple places in the genome and so a high enough value should be chosen to exclude these reads. The alignment score will vary between mappers and read length. A score of 76 would be perfect mapping, if the scoring scheme is 1 for a match and all the reads are 76bp long.

4.4 Any size genome, where only one library of reads is available

The most common case of one library only is a single short insert Illumina lane. In this case, use them as the 'long' insert reads as well as short. The commands are:

```
reapr perfectmap assembly.fa reads_1.fq reads_2.fq <insert_size> perfect
reapr smaltmap assembly.fa reads_1.fq reads_2.fq mapped.bam
reapr pipeline assembly.fa mapped.bam output_directory perfect
```

or for a large genome, use `perfectfrombam`, instead of `perfectmap`, as outlined above.

If only large insert reads are available, then it is probably not worth running `perfectmap`, since they are unlikely to be of high enough quality. You can still use REAPR to call errors, but won't get accurate error-free base calls. In this case, just run

```
reapr smaltmap assembly.fa reads_1.fq reads_2.fq mapped.bam
reapr pipeline assembly.fa mapped.bam output_directory
```

5 Interpreting the output

5.1 Sanity checks

It is a good idea to sanity check the output, to make sure that the input to the pipeline looks sensible. The first thing REAPR does is estimate various parameters by sampling from the reads. The results of this stage are in the directory `00.Sample/`. Things you should check:

- The GC/coverage bias - look at the plot `gc_vs_cov.lowess.pdf`.
- The fragment size distribution. There are three plots:
  `insert.in.pdf`, `insert.out.pdf` and `insert.same.pdf`.
  These show the distributions for the innies, outties and reads in the same direction. The innies plot should reflect your expected insert size distribution and the other two plots should just be noise.
- Check that the numbers in `insert.stats.txt` look sensible. See section §7.1 for more information on this file.

5.2 Summary file and errors/warnings GFF

As long as the sanity checks look OK, the next thing is to look at the final summary file `05.summary.report.txt`. This has the contiguity statistics of the input assembly, a summary of the errors and warnings made by REAPR and contiguity statistics of the assembly after breaking by REAPR. The four types of error are as follows.

1. FCD errors within a contig – a region thati does not contain a gap (one or more Ns) and triggered an FCD error
2. FCD error over a gap – as above, but the region does contain a gap
3. Low fragment coverage within a contig – a region with low fragment coverage (default is no coverage) that does not contain a gap.
4. Low fragment coverage over a gap – as above, but the region does contain a gap.

There are also warnings generated from other types of inconsistencies in the read mapping. The errors and warnings are written to the GFF file `03.score.errors.gff.gz`, which is indexed with `tabix`.

5.3 Broken assembly

REAPR makes a broken assembly, `04.break.broken_assembly.fa`, based on the error calls. If an error is over a gap, then that scaffold is broken into two at the gap. If the error contains more than one gap, then the scaffold is broken at each gap. If an error is not over a gap, then the region is replaced with Ns because usually it's not a breakpoint, but an insertion or deletion, and the contigs shouldn't be broken.

The sequences that are replaced with Ns are written either to the broken assembly file or to a separate file called `04.break.broken_assembly_bin.fa`. By default, any of these sequences that are shorter than 1000bp are written to the bin file, and the remaining longer sequences are written to the broken assembly. These sequences are given names the start with “REAPR_bin”. The 1000bp cutoff can be changed using the option `-m` when running `reapr break`.

If you want to be more agressive at this stage, by breaking at every error (regardless of whether or not it is over a gap), use the `-a` option. This can be a good tactic if you plan to run a scaffolder on the broken assembly, as any false-positive breaks should be fixed by the scaffolder anyway.

6 Common Tasks

The tasks that your are most likely to want to use are described below in detail.

6.1 Facheck

This checks that the names of the sequences in a FASTA file are compatible with the REAPR pipeline. It can also make a new FASTA file with safe names. Usage:

```
reapr facheck assembly.fa [new_assembly]
```

If the optional `new_assembly` is not given, this simply checks that the contig names in the FASTA file `assembly.fa` will not break the pipeline. It dies with an error message at the first occurrence of a bad name. If the optional `new_assembly` is given, then two files are output. The first is a new FASTA file, `new_assembly.fa`, with renamed contigs. The second is a file of old to new names, `new_assembly.info`, where each line is of the form

```
old_name	new_name
```

Fields are tab-delimited.

6.2 Smaltmap

This maps reads to the assembly using SMALT. The output is a sorted and indexed BAM file with duplicates removed. Usage:

```
reapr smaltmap [options] <assembly.fa> <reads_1.fastq> <reads_2.fastq> \
  <out.bam>
```

Options:

```
-k <int>
    The -k option (kmer hash length) when indexing the genome
    with 'smalt index' [13]
-s <int>
    The -s option (step length) when indexing the genome
    with 'smalt index' [2]
-m <int>
    The -m option when mapping reads with 'smalt map' [not used by default]
-n <int>
    The number of threads used when running 'smalt map' [1]
-y <float>
    The -y option when mapping reads with 'smalt map'.
    The default of 0.5 means that at least 50% of each read must map
    perfectly. Depending on the quality of your reads, you may want to
    increase this to be more stringent (or consider using -m) [0.5]
-x
    Use this to just print the commands that will be run, instead of
    actually running them
-u <int>
    The -u option of 'smalt sample'. This is used to estimate the insert
    size from a sample of the reads, by mapping every n^th read pair [1000]
```

6.3 Perfectmap

This is an optional first step of the pipeline, which is recommended if you have high quality paired reads. If generates parfect and uniquely mapping read coverage, for input into the REAPR pieline. Usage:

```
reapr perfectmap <assembly.fa> <reads_1.fastq> <reads_2.fastq> \
  <mean fragment size> <prefix of output files>
```

The n^{th} read in each fastq file should specify a read pair, with reads pointing towards each other. All reads must be the same length. If the FASTQ files have the extension `.gz`, they will be assumed to be gzipped and dealt with accordingly. The per-base coverage is written to a tab-delimited, bgzipped and tabix indexed file called `out.perfect_cov.gz`, where each line has the format

```
chromosome_name    position    coverage
```

and positions are 1-based. Note that any chromsomes that have zero coverage across their entire length will not be in this file. A histogram of the coverage is written to `out.hist`, with each line of the (tab-delimited) form

```
coverage    count
```

This file stops at coverage 100, with the count at coverage 100 meaning a coverage $\geqslant 100$.

This is very fast to run, but can by quite high memory, especially if the genome is large (more than a few 100MB). For large genomes, see `perfectfrombam`.

6.4 Perfectfrombam

This is an alternative to `perfectmap`, for use with large genomes. You will need a BAM file of paired reads as input, with each alignment record having an alignment score, `AS`, tag present. Usage:

```
reapr perfectfrombam [options] <in.bam> <prefix of output files> \
   <min insert> <max insert> <repetitive max qual> <perfect min qual> \
   <perfect min alignment score>
```

The BAM file will be filtered, so that only paired reads pointing towards each other, within the given insert size range and with at least the given mapping quality and alignment score are included. The filtered reads are used to generate the read depth across the genome. Output files are in the same format as those of `perfectmap`. Additionally, repetitive regions are called, by taking bases covered by any read with that read or its mate having mapping quality less than the cutoff chosen by 'repetitive max qual'.

6.5 Pipeline

This is a shortcut for running `facheck`, `preprocess`, `stats`, `fcdrate`, `score` and `break` in turn. See section 2 for examples. Usage:

```
  reapr pipeline [options] <assembly.fa> <in.bam> <out directory>  \
      [perfectmap prefix]
```

Options:

```
-stats|fcdrate|score|break option=value
    You can pass options to stats, fcdrate, score or break
    if you want to change the default settings. These
    can be used multiple times to use more than one option. e.g.:
        -stats i=100 -stats j=1000
    If an option has no value, use 1. e.g.
        -break b=1
-fcdcut <float>
    Set the fcdcutoff used when running score. Default is to
    run fcdrate to determine the cutoff. Using this option will
    skip fcdrate and use the given value.
-x
    By default, a bash script is written to run all
    the pipeline stages. Using this option stops the
    script from being run.
```

6.6 Plots

This makes plot files for a given sequence from the assembly, for viewing in Artemis. Prerequisites:

1. the per base statistics file made by `stats`;
2. the assembly in FASTA format.

Usage:

```
  reapr plots [options] <stats.per_base.gz> <out prefix> <assembly.fa> <contig id>
```

Options:

```
  -s, -score <scores prefix>
      This should be the outfiles prefix used when score was run
```

This will write several files to be read by Artemis. To start Artemis, run

```
  ./outprefix.run_art.sh
```

If you also ran `score`, then use the option `-s` to make a score plot and a GFF file for your contig of interest. These will also be opened when Artemis is started.

6.7 Seqrename

This can rename all the reference sequences in a BAM file. Use this if you already mapped your reads before finding that `facheck` failed. It saves spending the time remapping reads. Usage:

```
  reapr seqrename <rename file> <in.bam> <out.bam>
```

where `rename_file` is the `*.info` file made by `facheck`.

7 Advanced Tasks

The tasks listed in this section should not normally need to be used – they are for advanced users.

7.1 Preprocess

This stage samples from the assembly and BAM file in order to estimate various parameters such as fragment coverage. It also does the necessary calculations for GC vs coverage correction. Prerequisites:

1. A BAM file of paired reads mapped to the assembly. This BAM file needs to be sorted by coordinate (*e.g.* with `samtools sort`) and indexed (`samtools index`). It should also have duplicates marked (using `MarkDuplicates` from Picard `http://picard.sourceforge.net`) or removed (`samtools rmdup`). Note that the reads should point towards each other.

2. The assembly to which the reads were mapped, in FASTA format.

Usage:

```
reapr preprocess <assembly.fa> <in.bam> <outdir>
```

The files relating to the sampling are written in the directory `outdir/Sample/`. The estimates of insert size related statistics are written to `insert.stats.txt`. These should be self-explanatory, except `inner_mean_cov`, which is the mean coverage of *inner* fragments (often called the 'inner mate pair distance'). The `ave` value is the calculated 'average' insert size and is usually the mode insert size. However, particularly for large insert libraries, we often find that the mode is very small and not close to the mean. If this happens, then `ave` is set to the 'mode' within one standard deviation of the mean.

7.2 Stats

This step generates mapping statistics at each base of the assembly. Prerequisite: that you have run `preprocess`. Usage:

```
reapr stats [options] <preprocess output directory> <outfiles prefix>
```

Options:

```
-f <int>
    Insert size [ave from stats.txt]
-i <int>
    Minimum insert size [pc1 from stats.txt]
-j <int>
    Maximum insert size [pc99 from stats.txt]
-m <int>
    Maximum read length (this doesn't need to be exact, it just
    determines memory allocation, so must be >= max read length) [2000]
-p <string>
    Name of .gz perfect mapping file made by 'perfectmap'
-q <int>
    Ignore reads with mapping quality less than this [0]
-s <int>
    Calculate FCD error every n^th base
    [ceil((fragment size) / 1000)]
-u <string>
    File containing list of chromosomes to look at
    (one per line)
```

The main output file is `outprefix.per_base.gz`. This is a tab-delimited bgzipped and tabix indexed file containing statistics calculated at each base of the genome. Coordinates are 1-based. The columns of this file are described in Table 1.

In addition to the file of statistics, the following plots are made:

```
outprefix.read_coverage.pdf
outprefix.fragment_coverage.pdf
outprefix.fragment_length.pdf
outprefix.FCDerror.pdf
```

Each plot is generated using its associated `R` file `*.R`. The first two plots are histograms of the read and inner fragment coverage of each base of the assembly. The third plot is the distribution of fragment lengths. The final plot is the distribution of FCD errors at each base.

The `stats` task also writes a file of summary statistics, `outprefix.global_stats.txt`, which is described in Table 2.

7.3 Fcdrate

This calculates the cutoff to decide if a given window is an FCD failure or not. The cutoff is required to run the task `score`. Usage:

```
reapr fcdrate [options] <preprocess directory> <stats prefix> \
  <prefix of outut files>
```

Options:

```
  -l <int>
      Window length [insert_size / 2] (insert_size is taken to be
      sample_ave_fragment_length in the file global_stats.txt file made by stats)
  -p <int>
      Percent of bases in window > fcd cutoff to call as error [80]
  -s <int>
      Step length for window sampling [100]
  -w <int>
      Max number of windows to sample [100000]
```

Column number	Column heading	Description
1	`chr`	Sequence (chromosome/contig/scaffold) name
2	`pos`	Position in sequence (1-based)
3	`perfect_cov`	Proportion of perfect and uniquely mapping reads
4	`read_cov`	Read coverage on forwards strand
5	`prop_cov`	Proportion of properly paired reads on forwards strand
6	`orphan_cov`	Proportion of orphaned reads on forwards strand
7	`bad_insert_cov`	Proportion of reads with wrong insert size on forwards strand
8	`bad_orient_cov`	Proportion of reads in wrong orientation on forwards strand
9	`read_cov_r`	Read coverage on reverse strand
10	`prop_cov_r`	Proportion of properly paired reads on reverse strand
11	`orphan_cov_r`	Proportion of orphaned reads on reverse strand
12	`bad_insert_cov_r`	Proportion of reads with wrong insert size on reverse strand
13	`bad_orient_cov_r`	Proportion of reads in wrong orientation on reverse strand
14	`frag_cov`	Inner fragment coverage
15	`frag_cov_err`	Relative error in inner fragment coverage
16	`FCD_mean`	Mean insert size of just fragments covering this position, treating the plot as a histogram centred on zero (so a mean of zero is ideal)
17	`clip_fl`	Number of reads soft-clipped at the left end on forwards strand
18	`clip_rl`	Number of reads soft-clipped at the left end on reverse strand
19	`clip_fr`	Number of reads soft-clipped at the right end on forwards strand
20	`clip_rr`	Number of reads soft-clipped at the right end on reverse strand
21	`FCD_err`	FCD error. This is set to -1 if there are no fragments covering the position.
22	`mean_frag_length`	Mean length of the fragments covering this position

Table 1: Description of columns in the file `out.per_base.gz` made by the task `stats`.

Name	Description
`read_cov_mean`	Mean properly paired read coverage of each base
`read_cov_sd`	Standard deviation of properly paired read coverage
`read_cov_mode`	Mode properly paired read coverage
`fragment_cov_mean`	Mean inner fragment coverage
`fragment_cov_sd`	Standard deviation of inner fragment coverage
`fragment_cov_mode`	Mode inner fragment coverage
`fragment_length_mean`	Mean fragment size
`fragment_length_mean_sd`	Standard deviation of fragment size
`fragment_length_mode`	Mode fragment size
`fragment_length_min`	Minimum fragment size
`fragment_length_max`	Maximum fragment size
`use_perfect`	Whether or not perfect mapping reads were used (0 for no, 1 for yes)
`sample_ave_fragment_length`	Average fragment length from sample made by `preprocess`
`fcd_skip`	The skip distance for calculating the FCD error (every n$^{\text{th}}$ base is used)

Table 2: Description of columns in the file `out.global_stats.txt` made by the task `stats`.

The cutoff is determined by sampling windows across the genome. For each window, the cutoff needed to call this window as an error is calculated. In other words (using the defaults), for a given window we find the cutoff value c such that 80% of the values in this window are greater than c. This allows us to generate a plot of the proportion of windows which would be called as incorrect, for a range of FCD cutoff values. An example plot is shown in Figure 3. The FCD cutoff is chosen to be the first value encountered, starting from the right, such that both the first and second derivatives are more than 0.01. The derivatives are normalised so that they lie in the interval $[-1, 1]$. The idea is that we want to catch the turning point in the plot, to the left of which the majority of windows fail simply fail due to background noise.

7.4 Score

This task scores each base of the genome and reports errors in the assembly. The two most important files this task makes are as follows.

- `outprefix.score.gz`. This is a bgzipped, tab-delimited and tabix indexed file with lines of the form
  `chromosome_name   position    score`
  The score at each base ranges from -1 to 1. A score of 1 means no errors were detected. The smaller the score, the more errors were found, with 0 being the worst score. Bases within gaps are always given a score of -1.

- `outprefix.errors.gff.gz`. This is a bgzipped and tabix indexed `gff` file containing errors and warnings about the assembly. This file is required to run the task `break`. See Table 3 for a description.

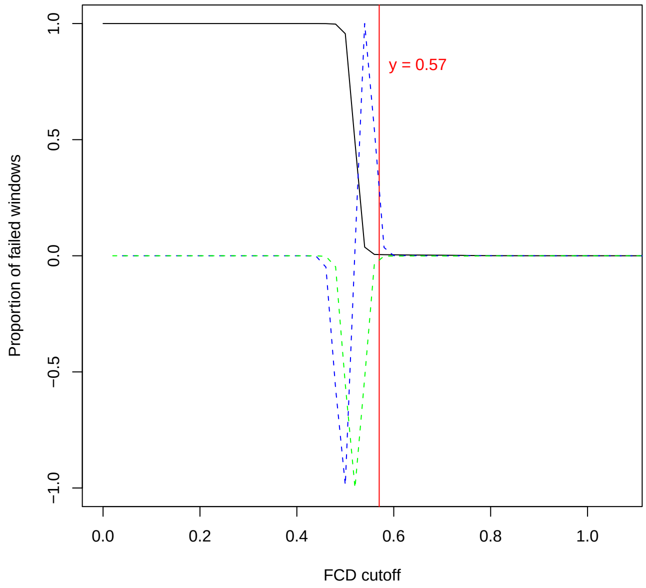

Figure 3: Example plot produced by the task `fcdrate`. The black line shows the proportion of failed windows at each cutoff value. The dashed green and blue lines are the first and second derivatives respectively of the black line. The read line shows the chosen cutoff value (in this example, at 0.57).

Prerequisites:

1. the same BAM file as was used as input to `stats`;
2. the statistics file made by `stats`.

Usage:

```
reapr score [options] <assembly.fa.gaps.gz> <in.bam> <stats prefix> \
  <FCD cutoff> <prefix of output files>
```

Options:

```
-f <int>
    Minimum inner fragment coverage [1]
-g <int>
    Max gap length to call over [0.5 * outer_mean_insert_size]
-l <int>
    Length of window [100]
-p <int>
    Use perfect mapping reads score with given min coverage.
    Incompatible with -P.
-P <int>
    Same as -p, but force the score to be zero at any position with
    at least the given coverage of perfect mapping reads and which has an
    OK insert plot, , i.e. perfect mapping reads + insert distribution
    override all other tests when calculating the score.
```

Error type	Description
FCD	This means that the FCD failed at the given coordinates. The score in the file (column 6) is the mean error in the region
FCD_gap	As for FCD, but there is a gap in the region
Frag_cov	The fragment coverage was too low in this region. The score in the file (column 6) is the mean fragment coverage in the region
Frag_cov_gap	As for Frag_cov, but there is a gap in the region
Low_score	Ths score in this region was too low (default $\leqslant 0.5$)
Link	This means that a significant proportion of the reads in this region mapped to elsewhere (to the region given in column 9) in the assembly
Clip	A significant proportion of the reads were clipped to map to this position (with reads clipped and mapped to both strands)
Repeat	A collapsed repeat, with the mean relative error in fragment coverage given in column 6
Read_cov	This region has low coverage of proper read pairs
Perfect_cov	Low coverage of perfect uniquely mapping reads in this region (default < 5)
Read_orientation	This region has a significant proportion of read pairs mapped in the wrong orientation (*i.e.* pointing away from each other or in the same direction)

Table 3: Description of error types in the file `out.errors.gff.gz` made by the task `score`. 'Error type' is what appears in the third column of the file.

```
    Incompatible with -p.
-q <float>
    Max bad read ratio [0.33]
-r <int>
    Min read coverage [max(1, mean_read_cov - 4 * read_cov_stddev)]
-R <float>
    Repeat calling cutoff. -R N means call a repeat if fragment
    coverage is >= N * (expected coverage).
    Use -R 0 to not call repeats [2]
-s <int>
    Min score to report in errors file [0.4]
-u <int>
    FCD error window length for error calling [insert_size / 2]
-w <float>
    Min % of bases in window needed to call as bad [0.8]
```

There are many options for this task, all of which can be calculated automatically from the results of `stats`. Each statistic, such as read coverage or fragment coverage, is analysed over a sliding window. If a window is found with a large fraction of errors, such as low fragment coverage, then that statistic as called as 'bad' in that window. The window length and cutoff values used to determine what constitutes a 'bad' score are all optional parameters, with defaults picked automatically.

7.5 Break

This task uses the errors file made by `score` to make a new version of the assembly, where scaffolds are broken at FCD or low fragment coverage errors. Regions called as an FCD error or have low fragment coverage, which do not contain a gap, are replaced with Ns. In case you want to keep these sequences, the regions are written to a separate FASTA file called `out_prefix.broken_assembly_bin.fa`.

Prerequisites:

1. the assembly in FASTA format;
2. the GFF errors file made by `score`.

Usage:

```
reapr break [options] <assembly.fa> <errors.gff.gz> <outfiles prefix>
```

Options:

```
-a
    Agressive breaking: break contigs at any FCD or low_frag error, as
    opposed to the default of replacing with Ns. Incompatible with -b
-b
    Ignore FCD and low fragment coverage errors that do not contain
    a gap (the default is to replace these with Ns). incompatible with -a
-e <float>
    Minimum FCD error [0]
-l <int>
    Minimum sequence length to output [1]
-m <int>
    Max sequence length to write to the bin. Sequences longer
    than this are written to the main assembly output. This is to stop
    long stretches of sequence being lost [999]
-t <int>
    When -a is used, use this option to specify how many bases
    are trimmed off the end of each new contig around a break.
    -t N means that, at an FCD error, a contig is broken at the middle
    coordinate of the error, then N bases are
    trimmed off each new contig end [0]
```

7.6 Summary

This is run at the end of the pipeline, produing a summary of the error calls and contiguity statistics of the original and broken assembly. Usage:

```
reapr summary [options] <assembly.fa> <score prefix> <break prefix> \
  <outfiles prefix>
```

where `score prefix` is the outfiles prefix used when score was run, and `break prefix`. is the outfiles prefix used when break was run. Options:

```
-e <float>
    Minimum FCD error [0]
```

8 References

[1] Derek W Barnett, Erik K Garrison, Aaron R Quinlan, Michael P Strömberg, and Gabor T Marth. BamTools: a C++ API and toolkit for analyzing and managing BAM files. *Bioinformatics (Oxford, England)*, 27(12):1691–2, June 2011.

[2] Tim Carver, Simon R Harris, Matthew Berriman, Julian Parkhill, and Jacqueline A McQuillan. Artemis: an integrated platform for visualization and analysis of high-throughput sequence-based experimental data. *Bioinformatics (Oxford, England)*, 28(4):464–9, February 2012.

[3] Heng Li. Tabix: fast retrieval of sequence features from generic TAB-delimited files. *Bioinformatics (Oxford, England)*, 27(5):718–9, March 2011.

[4] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, and Richard Durbin. The Sequence Alignment/Map format and SAMtools. *Bioinformatics (Oxford, England)*, 25(16):2078–9, August 2009.

[5] Heinrich Magnus Manske and Dominic P Kwiatkowski. SNP-o-matic. *Bioinformatics (Oxford, England)*, 25(18):2434–5, September 2009.

[6] R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2012. ISBN 3-900051-07-0.