Small RNA Pipeline

Steven Hill

July 16, 2013

1 Overview

The pipeline was constructed inside a tool created to manage each process. This tool is responsible for spawning each process and maintaining some sort of priority to the jobs. Being built in this format allowed for one process to easily run a job multiple times with different inputs and to execute several processes in parallel. This document will go over the pipeline manager and the processes associated with each step for the constructed pipeline.

The pre-constructed pipeline (smallrna.tar.bz2) requires several folders to be present before running. Inside the package you will find: the executables, demultiplexer, plotter, depth alignment, etc. They are located in the bin folder. There are symbolic links located in the root of the directory structure. The settings file can be found in settings.json. It contains the default setting for the pipeline and points to the test raw_ data folder.The folder also contains a makefile, this file has two commands, clean and stash. Stash will compress the entire folder into a tarball and then clean the directory. Clean will remove all the files and start from scratch.

2 Pipeline Manager

The pipeline manager requires one input, the settings file. The pipeline when run in verbose mode (-v, –verbose) will print out what was passed to standard error (stderr). It will also print a message that says ”Errors in program: PROGRAMNAME use -w for strict mode.” This does not mean that there is an error. It only means that something was written to stderr. Many bioinformatics programs print statistics to stderr.

The settings file (-s, –setting)is expected to be in json format and to meet the following format:

2.1 Settings File

```
{
        "Process Title" : {
                "path": "absolute/path/to/exe",
                "priority": 1,
                "arguments":
                        [
                                "arg1", "arg2"
                                "-flag", "assoc"
                        ]
        } ,

        "Process Title with Directory": {
                "path": "absolute/path/to/exe",
                "priority": 2,
                "directory": "/set/of/inputs",
                "arguments":
                        [
                                "inputexample" : "/set/of/inputs/$DIRECTORY",
                                "outputexample": "$DIRECTORY.output"
                        ]
        }
        "Process Title with Output": {
                "path": "absolute/path/to/exe",
                "priority": 3,
                "output": "myoutputs/output",
                "arguments": []
        }
}
```

2.2 Capturing Process Output

Many process output important information to standard error. This information is captured and written to pipeline_stats.txt. In this file you will find things like, percentage of adapters trimmed, percentage too long or too short. The percentage of alignments and many other things. For now, all this information is simply stored in the file. If specific information becomes a need, it is an easy step to add a parser to the pipeline and grab the important information.

2.3 Required Fields

The settings file is organized by processes. Each process starts with a title, this can be anything and will only be used during output. The required fields inside this object (following the curly brace), are path, priority, and arguments.

- path: expects a string which represents the path to executable
- priority: expects an integer which represents when the process will be executed. Processes with equal priority will be executed in parallel. Otherwise the smallest number will be executed first.
- arguments: expects a list of strings, these will be passed to the process as arguments. This field may be left blank by just inputting []

2.4 Optional Fields

In addition to the required fields, there are also two optional fields. Those are directory and output.

- directory: If the directory argument is present, the pipeline will create a copy of the process for each file in the directory (including priority), then it looks for any instance of the string $DIRECTORY and replace it with that file.
- output: The output argument expects a path to a file. It tells the application to pipe the output from stdout and place it into the file. This works exactly how the greater than operator works from a terminal.

2.5 Formatting

A common problem when executing the pipeline will be the format of the json file. If there is something wrong with the format, the pipeline will print an error message telling you approximately where the error occured.

Example:

```
Error parsing the settings file:
Expecting , delimiter: line 9 column 5 (char 130)
Exiting.
```

Common problems when working with json are:

- Missing commas between processes or argments
- Missing quotation to end field
- Colon provided where a comma should be placed, and vice-versa
- A comma at the end of a list, where there should be none

3 Demultiplexer

The demultiplexer is a script written in python that essentially looks for the provided barcodes and parses the input file into new files based on the barcode. The barcode is removed, and the new sequence is written along with the headers to the output file. There are two primary modes, one is to provide a R2 file which contains the barcodes for each line in addition to a barcode file, and the other is to just provide the barcode file. For the purpose of this pipeline we are using the first mode, which will check the barcode based on the sequence line in the R2 file. In addition, it checks that the headers are the same in each file, if they are different it will throw them into the unindexed bin.

When demultiplexing is complete, it will write to a file called stats.txt. This file will display the number of lines indexed, and the number of lines thrown out. Following the raw numbers, there is the percentage thrown out and the percentage kept.

By calling demultiplexer –help, it will display a usage message.

4 Intermediate steps

The following applications are common applications already installed on the cluster, they are used for intermediate steps and alignment.

4.1 Cutadapt

Cutadapt is responsible for cutting the adapter. It takes an argument for both the 3' adapter and the 5' adapter, however this is optional, and we only provide the 3' adapter, adding both adapters can cause for many reads to be cut short. Additionally, we can specify a minimum and maximum length, all other reads will be thrown out. It is important to note that cutadapt expects the input file as an argument with no flag.

4.2 Collapse

Fastx_collaper is used to collapse the application used to collapse reads that are the same. The only important thing to note is that -Q33 must be specified to work with our reads.

4.3 BWA Alignment

BWA aln is used to perform the alignment. We can pass this aligner the number of threads to use, and the settings file does so (16). Additionally, the database file and a query file must be provided as input. These are important as they will both be used again while working with samtools. The alignment output will be sent to stdout, so the pipeline uses the output field here.

4.4 Converting Alignment

The next step is to convert the output to sam format. The samtools application to do this is samse. The three primary arguments are the database file, the aligned output, and the query file. They must be in this order. All of samtools sends the output to standard out, so again the output field will be used in the settings file. After being converted to sam, they are again converted to bam, for easy use by the view command.

4.5 SAMTOOLS output

Finally, the files needed to be sorted based on if they are forward or reverse strand, additionally, reads that do not align must be thrown out. Samtools view will filter results based on the direction of the strand.

5 Plotter

The final step is to plot the depth of each alignment. The tool aligndepth will look through the sam file, and calculate the depth for each position in the genome. These depths are transformed by passing the value through the log10 function. It takes arguments of lengths, input, output, and minimum/maximum position. The input lengths will take a comma separated list of lengths and output each group into their own file. Aligndepth will then write to the file titled output and append _lengthM. IE, output_21M.

In the case of the preconstructed pipeline, 21,22,23,24 are provided with the –length argument. Minimum and maximum are not used, replotting the raw CSV files is more efficient and these arguments may be provided directly to the plotting tool. The preconstructed pipeline does this twice, for both forward and reverse reads.

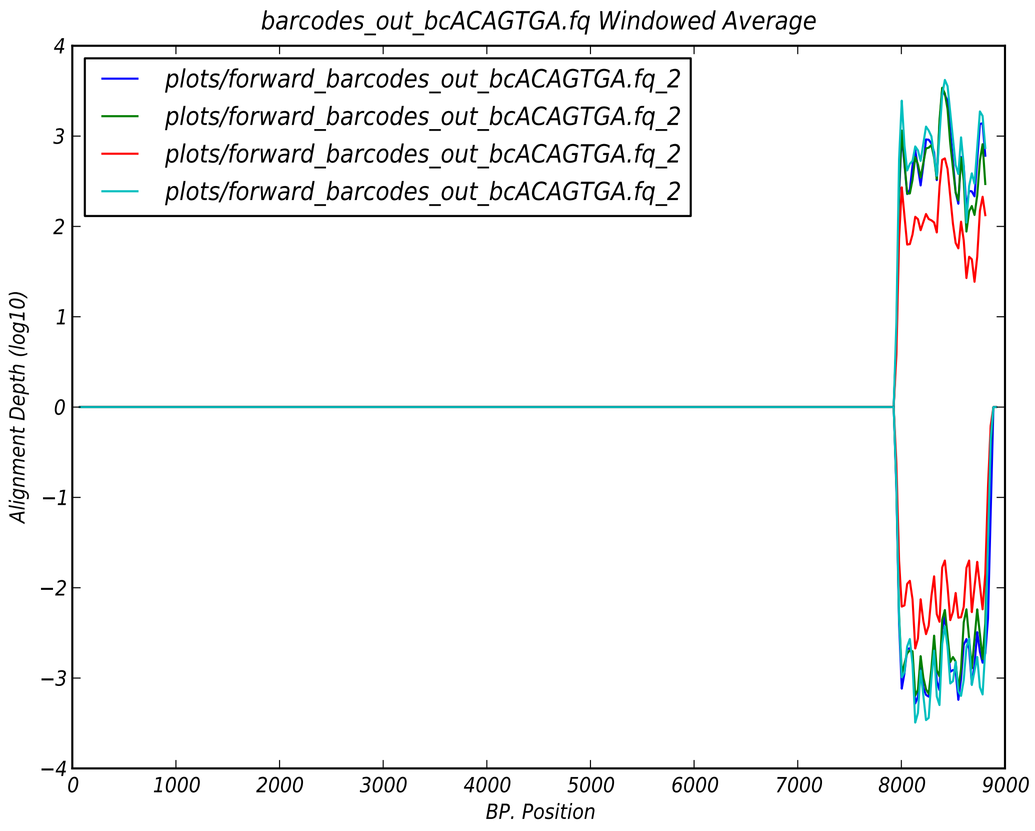
The next step is to use the actual plotting tool. The plotting tool, alignplot, supports a large amount of arguments for manipulating the plots. Since the plotter uses a windowed average to plot the different depths, the argument –step and –window are preconfigured. Step is set to 25, and window is set to 50. That is, the window contains 50 points, and each step it will shift by 25 points. This configuration provides a more normal configuration.

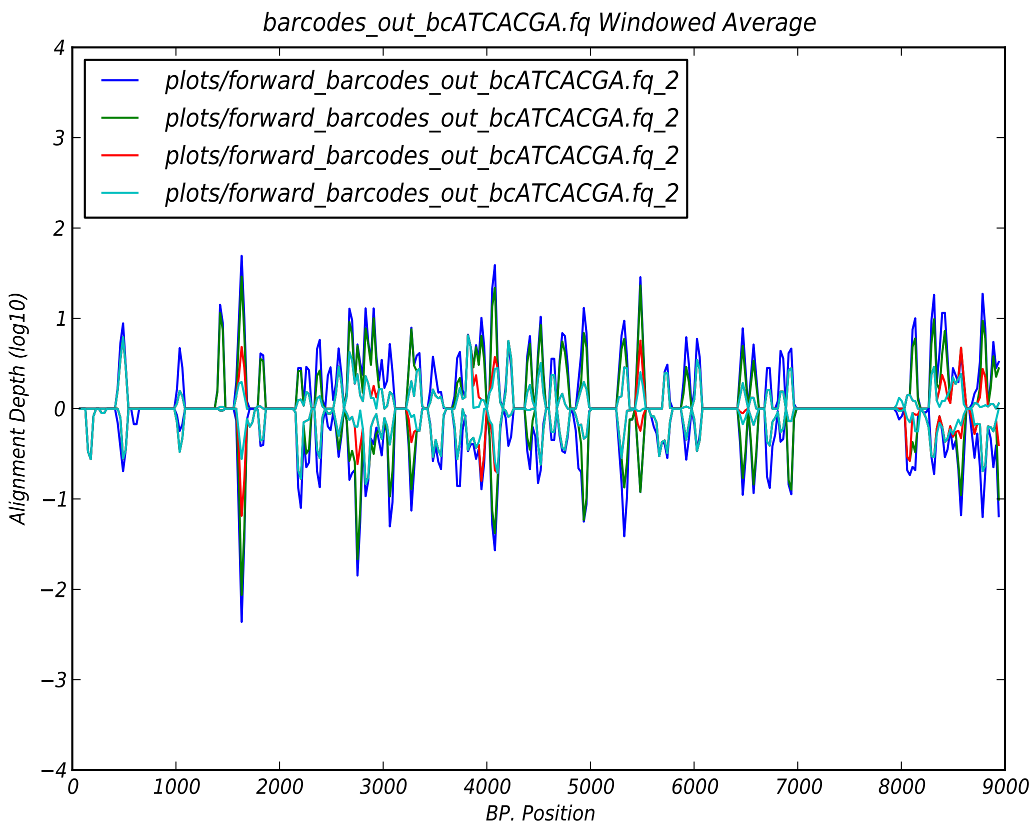
Additionally, the plotter expects two sets of input files in comma separated lists. Forward reads and reverse reads. Each input is given their own color, forward in the positive space and reverse in the negative space. The plotter also requires the output argument. This must contain the file extension as well, in the case of the preconstructed pipeline, it uses .pdf for the extension, and maintains the filename used previously.

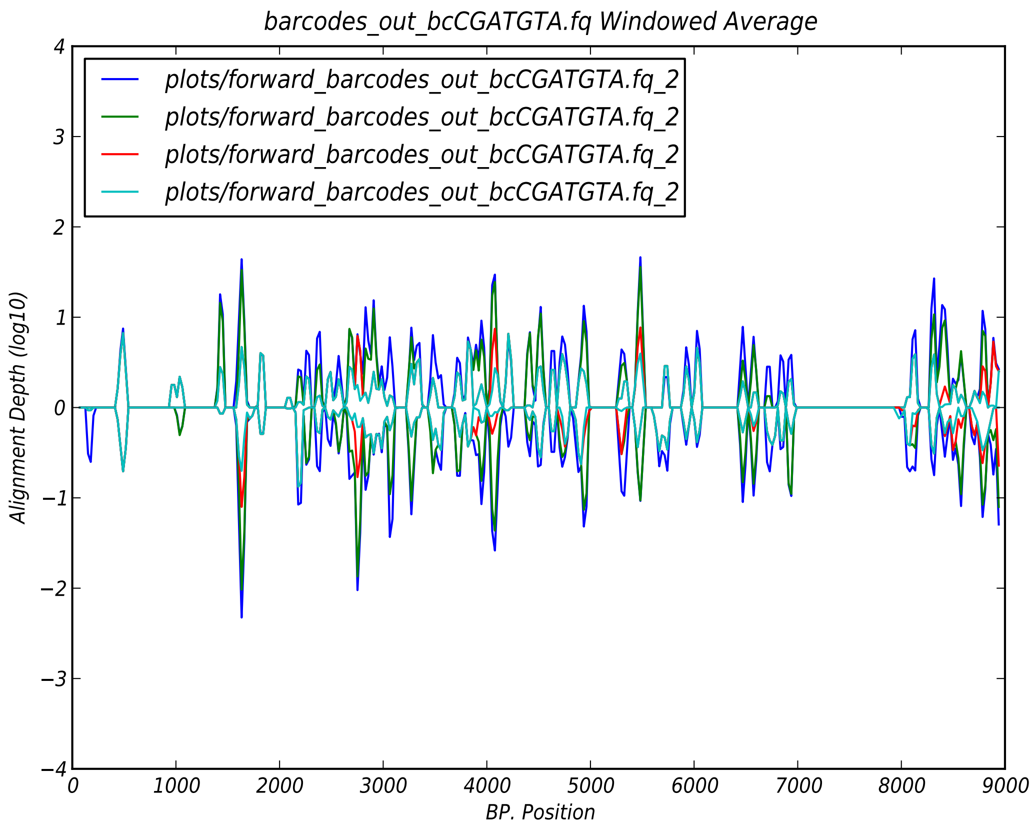
The plotter also supports optional arguments, such as Ymax, Ymin, minimum (xmin), maximum (ymax), and title.

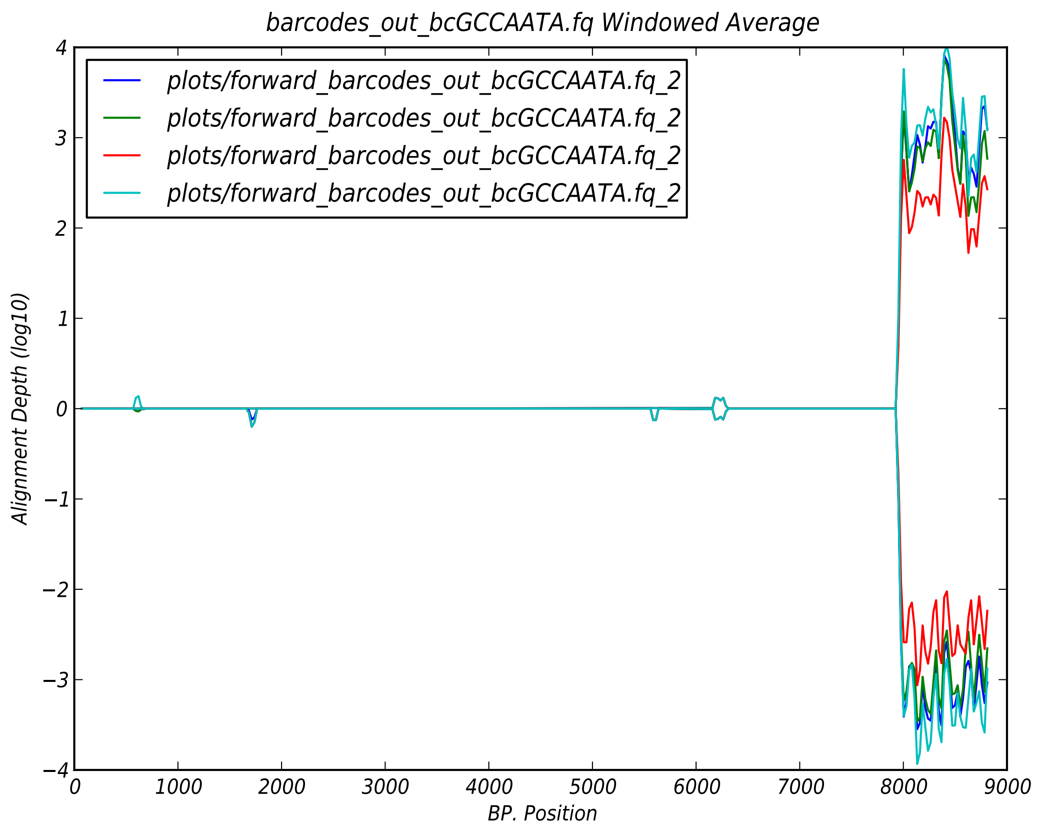
6 Plots

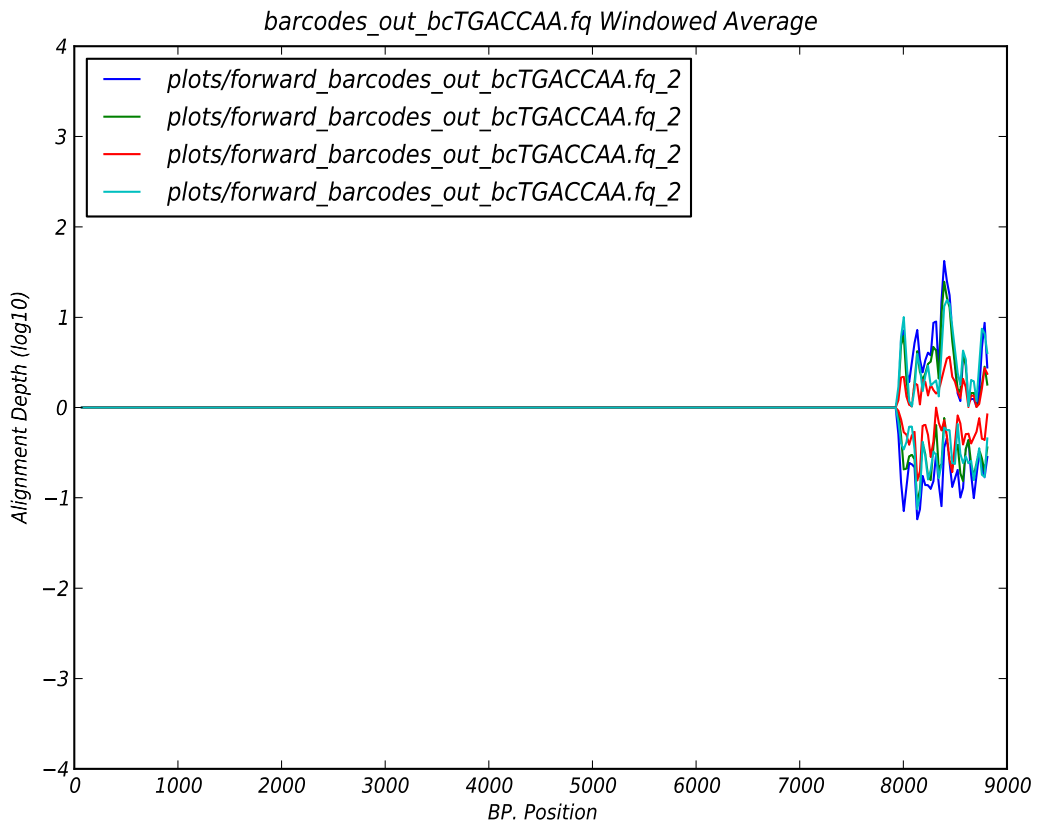
The following are plots generated by the pipeline on test data using the provided virus genome for aligment.

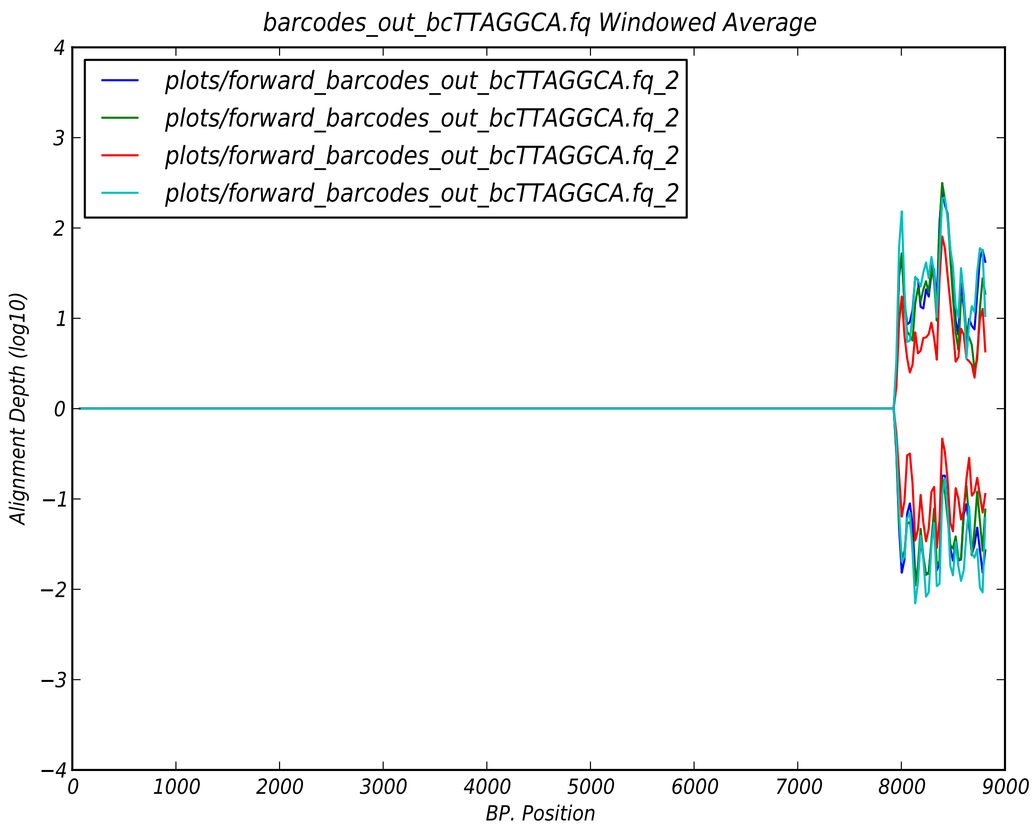



barcodes_out_bcATCACGA.fq Windowed Average
plots/forward_barcodes_out_bcATCACGA.fq_2
plots/forward_barcodes_out_bcATCACGA.fq_2
plots/forward_barcodes_out_bcATCACGA.fq_2
plots/forward_barcodes_out_bcATCACGA.fq_2
Alignment Depth (log10)
BP. Position
4
3
2
1
0
−1
−2
−3
−4
0
1000
2000
3000
4000
5000
6000
7000
8000
9000


barcodes_out_bcCGATGTA.fq Windowed Average
plots/forward_barcodes_out_bcCGATGTA.fq_2
plots/forward_barcodes_out_bcCGATGTA.fq_2
plots/forward_barcodes_out_bcCGATGTA.fq_2
plots/forward_barcodes_out_bcCGATGTA.fq_2
Alignment Depth (log10)
BP. Position
4
3
2
1
0
−1
−2
−3
−4
0
1000
2000
3000
4000
5000
6000
7000
8000
9000


barcodes_out_bcGCCAATA.fq Windowed Average
plots/forward_barcodes_out_bcGCCAATA.fq_2
plots/forward_barcodes_out_bcGCCAATA.fq_2
plots/forward_barcodes_out_bcGCCAATA.fq_2
plots/forward_barcodes_out_bcGCCAATA.fq_2
Alignment Depth (log10)
BP. Position
4
3
2
1
0
−1
−2
−3
−4
0
1000
2000
3000
4000
5000
6000
7000
8000
9000


barcodes_out_bcTGACCAA.fq Windowed Average
plots/forward_barcodes_out_bcTGACCAA.fq_2
plots/forward_barcodes_out_bcTGACCAA.fq_2
plots/forward_barcodes_out_bcTGACCAA.fq_2
plots/forward_barcodes_out_bcTGACCAA.fq_2
Alignment Depth (log10)
BP. Position
4
3
2
1
0
−1
−2
−3
−4
0
1000
2000
3000
4000
5000
6000
7000
8000
9000


barcodes_out_bcTTAGGCA.fq Windowed Average
plots/forward_barcodes_out_bcTTAGGCA.fq_2
plots/forward_barcodes_out_bcTTAGGCA.fq_2
plots/forward_barcodes_out_bcTTAGGCA.fq_2
plots/forward_barcodes_out_bcTTAGGCA.fq_2
Alignment Depth (log10)
BP. Position
4
3
2
1
0
−1
−2
−3
−4
0
1000
2000
3000
4000
5000
6000
7000
8000
9000