

BFAST: BLAT-like Fast Accurate Search Tool ¹

Nils Homer

¹Copyright © UCLA 2008, made for BFAST version 0.5.7

Contents

Table of Contents	iii
List of Figures	v
List of Tables	vii
Preface	ix
0.1 Acknowledgements	ix
1 Run BFAST now!	1
2 Fundamental Concepts	3
2.1 What is BFAST?	3
2.2 Work flow	4
3 Basic Usage	7
3.1 Common Options	7
3.1.1 Usage	7
3.2 bpreprocess	9
3.2.1 Creating a Reference Genome	9
3.2.2 Creating Indexes of a Reference Genome	9
3.2.3 Usage	10
3.3 bmatches	11
3.3.1 Finding Candidate Alignment Locations (CALs)	11
3.3.2 Usage	12
3.4 balign	14
3.4.1 Performing Local Alignment on Candidate Alignment Locations (CALs)	14
3.4.2 Usage	15
3.5 bpostprocess	17

3.5.1	Prioritizing Alignments	17
3.5.2	Usage	17
3.6	butil	18
3.6.1	bafconvert	18
3.6.2	balignmentscoredistribution	19
3.6.3	balignsim	19
3.6.4	berrordistribution	21
3.6.5	bevalsim	22
3.6.6	bexonify	22
3.6.7	bfixbaf	22
3.6.8	bfixbif	23
3.6.9	bfixbmf	23
3.6.10	bfixbrg	24
3.6.11	bfixhash	24
3.6.12	bgeneratereads	24
3.6.13	bheader	26
3.6.14	bindexdist	26
3.6.15	bindexhist	27
3.6.16	bmergesorted	27
3.6.17	bmfconvert	27
3.6.18	bpairedenddist	28
3.6.19	brepeat	28
3.6.20	brg2fasta	29
3.6.21	bsort	29
3.6.22	btestindexes	30
3.6.23	btranslocations	32
3.7	scripts	33
3.7.1	bfast.submit.pl	33
3.7.2	qseq2fastq.pl	33
3.7.3	solid2fastq	34
4	File Formats	35
4.1	Input Files	35
4.1.1	Reference genome FASTA file	35
4.1.2	Index Layout File	35
4.1.3	Exons File	36
4.1.4	BFAST index file file list	36
4.1.5	Reads FASTQ file	36

4.1.6	Offsets File	37
4.1.7	Scoring Matrix File	37
4.2	BFAST Files	37
4.2.1	BFAST reference genome file	38
4.2.2	BFAST index file	38
4.2.3	BFAST matches file	39
4.2.4	BFAST reads filtered file	39
4.2.5	BFAST aligned file	39
4.2.6	BFAST not aligned file	40
4.2.7	BFAST reported file	40
4.2.8	BFAST not reported file	40
4.2.9	BFAST multiple sequence alignment file	41
4.3	Example Input Files	41
5	Advanced Topics	47
5.1	How To Design Indexes	47
5.2	Whole-Genome Alignment	49
5.3	Targeted Genomic Alignments	49
5.3.1	Using bpreprocess and exon list	49
5.3.2	Using bexonify on existing indexes	49
5.3.3	Using command-line options to specify one contiguous range	49
5.4	Transcriptome Alignment	50
5.5	Bisulfite Treated or Methylation Alignment	51
5.6	Color Space Alignment	51
6	Examples	55
6.1	Whole-Genome Alignment of Phi X 174	55
7	Appendix	59
7.1	Human Genome Alignment Recommended Settings	59
7.1.1	Illumina	59
7.1.2	ABI SOLiD	61
7.2	High-Speed Tutorial	62
7.3	Copyright	62

List of Figures

2.1	The BFAST work flow.	6
4.1	An example of a reference genome FASTA file.	42
4.2	An example of an Index Layout file.	43
4.3	An example of an Exons file	43
4.4	An example of a BFAST index file list file.	44
4.5	An example of a reads FASTQ file in nucleotide space.	44
4.6	An example of a reads FASTQ file in nucleotide space with paired end reads.	45
4.7	An example of a reads FASTQ file in color space.	45
4.8	An example of an Offsets file.	46
4.9	An example of a Scoring Matrix file for nucleotide space.	46
4.10	An example of a Scoring Matrix file for color space.	46
5.1	The BFAST work flow for color space alignment.	53

List of Tables

Preface

This document is meant to serve as a guide for the practical use of BFAST. It includes explanations of all command-line options for each binary in BFAST to give an idea of basic usage. Input and output file formats are also detailed. We give many examples of use, including alignment to the whole genome, targeted genomic regions, and to the transcriptome.

This document does not try to explain the underlying algorithm or data-structures used in BFAST. For example, we will not explain the local alignment algorithms, or the underlying BFAST algorithm. Without proper understanding of the underlying algorithm, including local alignment algorithms, it is difficult to use this very flexible program knowledgeably to obtain your desired results. Therefore, reading the BFAST paper (?) is a strict prerequisite. Additionally, the local alignment paper used by BFAST for ABI SOLiD data is required for ABI SOLiD data alignment (?).

If you have anything that you would be useful to add to this guide, feel free to relay the addition to the BFAST developers. This includes but is not limited to bugs, typos, and explanations. Please see <http://genome.ucla.edu/bfast> for more details.

Enjoy!

0.1 Acknowledgements

We would like to thank Jim Kent for creating and distributing the BLAT program (?), which was both a motivation for this work, and was also the basis for our initial practical large scale alignment pipeline. We also thank members of the Nelson Lab: Zugen Chen, Hane Lee, Bret Harry, Jordan Mendler, Brian OConnor for input and computational infrastructure support.

The layout and organization of this book was guided by the design found in the SVN book (<http://svnbook.red-bean.com/>). We would also like to thank SVN for creating amazing version control software that manages the source code BFAST.

Chapter 1

Run BFAST now!

So impatient! If you want to run BFAST now, you have come to the right place. If you are aligning Illumina data or ABI SOLiD data to the human genome, please see [section 7.1](#). Otherwise, then please check out the High Speed Tutorial ([section 7.2](#)).

Chapter 2

Fundamental Concepts

2.1 What is BFAST?

The short answer is that BFAST is a tool for fast and accurate sequence alignment (it is in the name). An implementation of BFAST can be found at <http://genome.ucla.edu/bfast>. The longer answer is, well, longer.

The basic problem is as follows. We are given some reference sequence, to which we wish to compare our data. Our data are short reads (sometimes numbering in the billions) that are substrings of the reference sequence. In some cases, these short reads differ based on various edit operators that modify the read in some fashion. It is the goal of BFAST to find the location in the reference sequence that minimizes the number of edit operators used to transform the read into the reference sequence at that location. The process of concluding a read has a location in the reference sequence is called mapping or alignment.

In the case of DNA sequence alignment, these edit operators are biologically and technology motivated, including mutations/mismatches/SNPs, insertion of bases (or letters), deletion of bases (or letters), and technical errors. Typically we have some set of sequence to which we wish to compare our reads. In many cases, finding the correct location given a set of variants (such as the operators) is extremely important since we may be only interested in the variants themselves.

For example, we may be interested in mutations in Breast Cancer and therefore mapping accurately or robust to many combinations of variants is important. In other cases, the speed of mapping is important and the accuracy is not. For example, in digital gene expression analysis the goal is to estimate the number of reads map over locations in the genome as to gain an overall trend of coverage, rather than precise variant detection.

Whatever is the intended application of BFAST, it can be tuned to find alignments quickly at the cost of accuracy or robustness against variants. It can also be tuned to accurately find

alignments with variants at the cost of speed. This is the power of BFAST. It is a highly generalized algorithm allowing the user to tune the algorithm to suit their specific needs.

Its implementation is ever-evolving, with new releases occurring early in development and often. BFAST is suited for use on high memory machines when used for Human Genome resequencing and is best used on a cluster of such machines. It also uses multi-threaded programming to utilize the multi-core architecture of current machines. The output of BFAST is in binary format to reduce file space for large data, although we supply tools to convert these formats to text if so desired. Finally, we use the C programming language to implement BFAST to increase efficiency as well as the GNU Build System (Autotools) to handle compilation.

2.2 Work flow

The basic work flow has five steps as seen in [Figure 2.1](#).

1. The first step is to create a reference genome. This reference genome contains all the sequence to which we wish to align. The binary `bpreprocess` performs this task (see [section 3.2](#)).
2. The second step is to create indexes of the reference genome, which was created in the first step. The number and layout of these indexes is determined both by the user's speed and accuracy requirements. The binary `bpreprocess` performs this task (see [section 3.2](#)).
3. The third step is to find candidate alignment locations (CALs) for each read. The expected number of CALs returned is a function of the number of indexes and the layouts chosen in the second step as well as the number offsets. The binary `bmatches` performs this task (see [section 3.3](#)).
4. The fourth step is to fully align each CAL for each read. This uses a standard local alignment algorithm (?) or a custom tool for ABI SOLiD data (?). The binary `balign` performs this task (see [section 3.4](#)).
5. The fifth and final step is to prioritize the final alignments. The user specifies criteria to select the correct alignment for each read. The criteria can be based on many factors, including uniqueness, score, or other factors. The binary `bpostprocess` performs this task (see [section 3.5](#)).

The reference genome (Step 1) and indexes (Step 2) can be re-used between experiments where only the read data changes, not the reference genome or index layouts. Since only

the reads will change, one can utilize the same reference genome and indexes created in Step 1 and Step 2 respectively. Therefore, it is advised to perform Step 1 and Step 2 such that the reference genome and indexes can be re-used, thus reducing the work flow down to three steps upon re-use. In fact, the main speed of this program comes from the idea that in general, the reference genome and associated indexes need only be built once, with the reads coming from the same reference genome (say Human) but different samples or experiments.



Figure 2.1: The BFAST work flow.
See [section 2.2](#) for a description.

Chapter 3

Basic Usage

BFAST is a command-line program. It accepts many command-line options to customize and tune the alignment algorithm. As seen in the [section 2.2](#), there are five binaries to be run for the entire work flow, although only three need to be used if we are aligning to the same reference genome using the same indexes.

3.1 Common Options

Some common options exist across some or all of the binaries. These options include specifying the number of threads for parallel processing (`-n`), the number of reads to load at a time (`-Q`), naming the output files (`-o`), specifying where the output files should be created (`-d`), specifying where temporary files should be stored (`-T`), outputting timing information (`-t`), printing program parameters (`-p`), and printing a help message (`-h`).

Other options, such as `-A` for specifying the encoding space and the options `-s`, `-S`, `-e`, and `-E` for specifying only a contiguous range should be considered, are shared across some of the binaries but have specific implications to each binary and are described in the respective binary's section.

3.1.1 Usage

`-n INTEGER, --numThreads=INTEGER`

This option applies to `bpreprocess` (with option `-a 1`), `bmatches`, and `balign`. For `bpreprocess` (with option `-a 1`) the number of threads must be a power of two due the implementation of the index sorting algorithm (merge sort). Otherwise it is recommended that the number of threads match the number of cores or processors.

`-Q INTEGER, --queueLength=INTEGER`

Specifies the number of reads to cache or load into memory at one time.

`-o STRING, --outputID=STRING`

This option applies to `bpreprocess`, `bmatches`, `balign`, and `bpostprocess`. The given string (no spaces) will be inserted into the file name of each output file created by the given binary.

`-d DIRECTORY --outputDir=DIRECTORY`

This option applies to `bpreprocess`, `bmatches`, `balign`, and `bpostprocess`. This option specifies the output directory to which all files will be outputted. Be sure to include a trailing backslash or `\`.

`-T DIRECTORY --tmpDir=DIRECTORY`

This option applies to `bpreprocess` (with option `-a 1`), `bmatches`, and `balign`. This option specifies the directory in which to store temporary files. For large datasets, the necessary disk space for temporary files may be large and therefore it is useful to specify the temporary file directory. Be sure to include a trailing backslash or `\`. If no option is given, the temporary file directory is defaulted to the output directory (`-d`).

`-t, --timing`

This option applies to `bpreprocess`, `bmatches`, `balign`, and `bpostprocess`. This option causes timing information for the execution of the program to be displayed upon successful termination.

`-p, --Parameters`

This option applies to `bpreprocess`, `bmatches`, `balign`, and `bpostprocess`. This option causes the input command-line parameters to be displayed and subsequent termination of the program.

`-h, --Help`

This option applies to `bpreprocess`, `bmatches`, `balign`, and `bpostprocess`. This option prints a help message.

3.2 bpreprocess

bpreprocess is a command-line utility that is used to create a reference genome and indexes of a reference genome. This utility performs the first and second step of the work flow outlined in [section 2.2](#). Both the BFAST reference genome file and BFAST index file will be written in binary format to preserve space. See [subsection 4.2.1](#) for the file format of the BFAST reference genome file and see [subsection 4.2.2](#) for the file format of the BFAST index file.

Also see [section 3.1](#) for common options that are in use across some or all of the binaries.

3.2.1 Creating a Reference Genome

To create a reference genome, the required command-line options are `-r` and `-a`.

To specify reference genome creation, use the option `-a 0`. We need to input a reference genome FASTA file using the `-r` option (see [subsection 4.1.1](#) for the file format). Finally, to create a reference genome in color space, we use the option `-A 1`, otherwise we use `-A 0`.

When creating a BFAST reference genome file, the contigs will be numbered according to their order in the reference genome FASTA file (option `-r`). The numbering is one-based (begins with one). The maximum number of contigs is 2^{31} or 2147483648. The name of each contig specified in the header of the reference genome FASTA file will be also be stored. The maximum sequence length for a single contig is also 2^{31} or 2147483648. The input sequence will be assumed to be the forward strand of the genome. Only the forward strand of the genome will be stored (see [subsection 4.2.1](#) for more details).

The output will be a BFAST reference genome file (see [subsection 4.2.1](#) for the file format).

Other options are specified in [subsection 3.2.3](#).

3.2.2 Creating Indexes of a Reference Genome

To create indexes of a reference genome, the required command-line options are `-r`, `-a`, and `-i`.

To specify index creation, use the option `-a 1`. We input the BFAST reference genome file, which was created previously, using the `-r` option. To create the indexes in color space, we use the option `-A 1`, otherwise we use `-A 0`. If we choose to create the indexes in color space, the BFAST reference genome file must have been created in color space. Finally, we give the layouts for the indexes in a file (see [subsection 4.1.2](#) for the file format) using the `-i` option. The design of indexes is discussed in [section 5.1](#).

When creating a BFAST index file, we will use all possible contig sequences from the specified BFAST reference genome file, specified using option `-r`, unless any of the options

-s, -S, -e, -E, or -x are used. The file size of the BFAST index file can be very large for large indexes (see [subsection 4.2.2](#) for more details).

The output will be a BFAST index file (see [subsection 4.2.2](#) for the file format).

Other options are specified in [subsection 3.2.3](#).

3.2.3 Usage

`-r FILENAME, --rgFileName=FILENAME`

If we are creating a BFAST reference genome file (`-a 0`), then the file name should contain the reference genome FASTA file (see [subsection 4.1.1](#) for the file format). If we are creating a BFAST index file (`-a 1`), then the file name should contain the BFAST reference genome file (see [subsection 4.2.1](#) for the file format).

`-a INTEGER, --algorithm=INTEGER`

To create a BFAST reference genome file use `-a 0`. To create a BFAST index file use `-a 1`.

`-A INTEGER, --space=INTEGER`

Specifies the encoding space of the alphabet. For nucleotide space, use `-A 0`. For color space, use `-A 1`. If we are creating a BFAST index file, the space must be the same as the BFAST reference genome file used for input with option `-r`.

`-i FILENAME, --indexLayoutFilename=FILENAME`

Specifies the file that contains the layout(s) for the index(es) that you wish to create (for use with `-a 1`). To create multiple indexes in serial, include multiple entries in this file. The indexes will be numbered according to their order in the Index Layout file. For more information on this file format, see [subsection 4.1.2](#).

`-R, --RepeatMasker`

Ignores lower case bases when creating the indexes (`-a 1`). This typically corresponds to RepeatMasker sequence.

`-s INTEGER, --startContig=INTEGER`

Specifies the first contig to include when building indexes.

`-S INTEGER, --startPos=INTEGER`

Specifies the first position in the first contig to include when building indexes.

`-e INTEGER, --endContig=INTEGER`

Specifies the last contig to include when building indexes.

`-E INTEGER, --endPos=INTEGER`

Specifies the last position in the last contig to include when building indexes.

`-x FILENAME, --exonsFileName=FILENAME`

Specifies the exon-like ranges to include in the index. This option cannot be used with the `-s`, `-S`, `-e`, or `-E` options. The exon ranges must fall within bounds in the BFAST reference genome file. For the file format of the exons file, please see [subsection 4.1.3](#).

3.3 **bmatches**

bmatches is a command-line utility that takes a set of reads and searches a set of indexes to find candidate alignment locations (or CALs) for each read. This utility performs the third step in the work flow outlined in [section 2.2](#).

The output will be a BFAST matches file (see [subsection 4.2.3](#) for the file format) and a BFAST reads filtered file (see [subsection 4.2.4](#) for the file format).

Also see [section 3.1](#) for common options that are in use across some or all of the binaries.

3.3.1 Finding Candidate Alignment Locations (CALs)

To find CALs for a set of reads, the required command-line options are `-r`, `-i`, and `-R`. The `-r` option specifies the BFAST reference genome file. The `-i` option specify the file containing the list of file paths to the main BFAST reference genome files respectively (see [subsection 4.1.4](#)). If you wish to have a secondary set of indexes, which are used if no matches are found in the main set of indexes, use the `-I` option. If the option `-A 1` is used, then both the BFAST reference genome file and the BFAST index files must have been created using the `-A 1` option. The file containing the reads is specified using the `-R` option (see [subsection 4.1.5](#) for the file format).

There are two output files. The first output file is a BFAST matches file, which stores the CALs for each read in a binary format (see [subsection 4.2.3](#) for the file format). The

second output file is a BFAST reads filtered file, which stores those reads that were initially filtered (see [subsection 4.2.4](#) for the file format).

Next, CALs will be found for each read using the main BFAST index files using the `-i` option. If no CALs were found in the main BFAST index files for a given read, then CALs will be found using the secondary BFAST index files using the `-I` option.

Only the forward strand of the genome is indexed (see [subsection 4.2.2](#)), so both the read and its reverse complement will be looked-up in the index to find CALs. This can be modified by using the `-w` option, which will target a specific strand of the reference genome.

In all cases, the BFAST reference genome file and BFAST index files must match that encoding space specified by `-A`.

The `-K` and `-M` options are useful to ignore keys that return too many CALs (`-K`) and to ignore reads that in aggregate have too many CALs (`-M`).

If the reads file is large, a subset of reads can be specified using the `-s` and `-e` options, which helps distribute the process across a cluster. For extremely large read datasets (billions), it is recommended that the reads be split into separate files before hand.

Other options are specified in [subsection 3.3.2](#)

3.3.2 Usage

`-r FILENAME, --rgFileName=FILENAME`

This option specifies the BFAST reference genome file (see [subsection 4.2.1](#) for the file format).

`-i FILENAME, --bfastMainIndexesFileName=FILENAME`

Specifies the file containing the paths to the main BFAST index files. See [subsection 4.1.4](#) for this file format. See [subsection 4.2.2](#) for the file format of the BFAST index files.

`-I FILENAME, --bfastSecondaryIndexesFileName=FILENAME`

Specifies the file containing the paths to the secondary BFAST index files. See [subsection 4.1.4](#) for this file format. See [subsection 4.2.2](#) for the file format of the BFAST index files. To use no secondary indexes, given an empty file as input.

`-R FILENAME, --readsFileName=FILENAME`

Specifies the file containing the reads. See [subsection 4.1.5](#) for more information on the file format of the reads file.

`-O FILENAME, --offsetsFileName=FILENAME`

Specifies the file containing the offsets to use for all BFAST index files. See [subsection 4.1.6](#) for more information on the file format of the Offsets file.

`-A INTEGER, --space=INTEGER`

Specifies the encoding space of the alphabet. For nucleotide space, use `-A 0`. For color space, use `-A 1`. When using this option, the BFAST reference genome file and BFAST index files must also have been created in the same encoding space. That is, they must also have been created using the same `-A` option.

`-s INTEGER, --startReadNum=INTEGER`

Specifies the first read in which to process. This may be useful when distributing a large data set across a cluster.

`-e INTEGER, --endReadNum=INTEGER`

Specifies the last read in which to process. This may be useful when distributing a large data set across a cluster.

`-k INTEGER, --keySize=INTEGER`

Specifies to truncate all indexes to have the given key size. This will only be performed on indexes for which the given value is greater than the hash width and less than the original key size. This may be useful to search with greater sensitivity by reusing indexes large key sizes.

`-K INTEGER, --maxKeyMatches=INTEGER`

Specifies the maximum number of matches to allow before a key is ignored. A key may return one or more CALs and therefore it may be desirable to ignore non-unique or over-represented keys. For example, a value of 100 may be useful when aligning the to Human Genome given that each index used is expected to return one CAL.

`-M INTEGER, --maxNumMatches=INTEGER`

Specifies the maximum number of CALs to allow before we stop searching for CALS for a given read. If the limit is reached, the read will be flagged and ignored in later alignment

processes. For example, a value of 500 may be useful when aligning the to Human Genome given that each index used is expected to return one CAL.

`-w INTEGER, --whichStrand=INTEGER`

Specifies to find matches on the designated strands. For both strands, use `-w 0`. For the forward strand only, use `-w 1`. For the reverse strand only, use `-w 2`.

3.4 balign

balign is a command-line utility that takes a list of Candidate Alignment Locations (CALs) for each read and performs a local alignment of each read to the reference, giving a score for the quality of the alignment. This utility performs the fourth step in the work flow outlined in [section 2.2](#).

The output will be a BFAST aligned file (see [subsection 4.2.5](#) for the file format) and a BFAST not aligned file (see [subsection 4.2.6](#) for the file format). Also see [section 3.1](#) for common options that are in use across some or all of the binaries.

3.4.1 Performing Local Alignment on Candidate Alignment Locations (CALs)

To perform local alignment of CALs, the required options are `-r` and `-m`. The `-r` option specifies the reference genome FASTA file, which for this input must always have been created with nucleotide space encoding. The `-m` option specifies the BFAST matches file outputted by `bmatches`.

There are two output files. The first output file is a BFAST aligned file, which stores the local alignments for each CAL and read in binary format (see [subsection 4.2.5](#) for the file format). The second output file is a BFAST not aligned file, which stores those reads for which local alignment was not performed (see [subsection 4.2.6](#) for the file format). Local alignment may be time consuming when a large number of CALs are returned. Therefore we can use the option `-M` to specify maximum number of CALs. If a read has more than the specified number, it will be ignored, and printed to the BFAST not aligned file. Additionally, reads with no CALs will also be outputted to the BFAST not aligned file.

The Smith Waterman algorithm supports mismatches, indels (with affine gap penalties), and color errors. The value for the `-A` option must match the value given to the `-A` option in `bmatches`. In the case that `-A 1` is used, the algorithm will simultaneously correct for color errors. For more information on the local aligner, please see [?](#).

The `-s`, `-S`, `-e` and `-E` can be used to specify only to consider CALs within a given range.

For any paired end read using option `-l` will infer a set of CALs for one read using the CALs from the second read, given the first read has no CALs. The expected length between the reads should be given as input to the option. We can use the `-L` option to mirror in various directions ($5' \rightarrow 3'$ or $3' \rightarrow 5'$) according the underlying sequencing technology used. If we wish to force this inference for all reads, not just those that have one read without CALs and the other with CALs, then the option `-f` should be specified.

3.4.2 Usage

`-r FILENAME, --rgFileName=FILENAME`

This option specifies the BFAST reference genome file (see [subsection 4.2.1](#) for the file format). Regardless of option `-A`, the BFAST reference genome file should have been created in nucleotide space.

`-m FILENAME, --matchFileName=FILENAME`

Specifies the BFAST matches file outputted by the `bmatches` utility. See [subsection 4.2.3](#) for the file format.

`-x FILENAME, --scoringMatrixFileName=FILENAME`

Specifies the Scoring Matrix file used to score the alignments. Please see [subsection 4.1.7](#) for the file format.

`-a INTEGER, --alignmentType=INTEGER`

Specifies the type of local alignment to perform. `-a 0` specifies that full local alignment should be performed, where we consider mismatches, insertions, deletions and color errors (if `-A 1` is specified). `-a 1` specifies that only local alignment considering mismatches should be performed and color errors (if `-A 1` is specified).

`-A INTEGER, --space=INTEGER`

Specifies the encoding space of the alphabet. For nucleotide space, use `-A 0`. For color space, use `-A 1`, which will use a modified Smith Waterman algorithm to correct for color space errors.

`-s INTEGER, --startReadNum=INTEGER`

Specifies the first read in which to process. This may be useful when distributing a large data set across a cluster.

`-e INTEGER, --endReadNum=INTEGER`

Specifies the last read in which to process. This may be useful when distributing a large data set across a cluster.

`-O INTEGER, --offset=INTEGER`

Specifies the number of bases before and after each CAL to include in the reference when aligning. For example, a value of 10 can be used when aligning to the Human Genome, since this would allow for small insertions and deletions to be placed more accurately in the local alignment.

`-M INTEGER, --maxNumMatches=INTEGER`

Specifies to ignore reads who have more than the specified number of CALs.

`-q INTEGER, --avgMismatchQuality=INTEGER`

Specifies the average mismatch quality.

`-l INTEGER, --pairedEndLength=INTEGER`

Specifies that if one read of the pair has CALs and the other does not, this distance will be used to infer the latter reads CAL.

`-L INTEGER, --mirroringType=INTEGER`

Specifies the directionality of the mirroring. A value of `-f 0` that no mirroring should occur. A value of `-f 1` specifies that we assume that the first end is before the second end ($5' \rightarrow 3'$). A value of `-f 2` specifies that we assume that the second end is before the first end ($5' \rightarrow 3'$). A value of `-f 3` specifies that we mirror CALs in both directions.

`-f INTEGER, --forceMirroring INTEGER`

Specifies that we should always mirror CALs using the distance specified in option `-l` for paired end reads, even if both of the reads have CALs.

3.5 bpostprocess

bpostprocess is a command-line utility that takes as input a BFAST aligned file. It can convert the input file to a specified output format as well as help choose the best alignment for each read based on score or uniqueness. This utility performs the fifth step in the work flow outlined in [section 2.2](#). Many other filters can be applied, for example with paired-end reads where we desire only alignments for which BOTH ends align. These filters can be applied by downstream tools such as SAMtools (see <http://samtools.sourceforge.net>).

There are two output files, a BFAST reported file (see [subsection 4.2.7](#) for the file format) and BFAST not reported file (see [subsection 4.2.8](#) for the file format). The BFAST reported file can be in three formats depending on the desired output format using the option `-O`.

Also see [section 3.1](#) for common options that are in use across some or all of the binaries.

3.5.1 Prioritizing Alignments

The `-a` option can be used to filter and choose the best alignment. `-a 0` will not modify the data but only convert the file to the specified output format (`-O`). Options `-a 1`, `-a 2`, and `-a 3`, will for each read select a subset of alignments from the alignment(s) found. `-a 1` will output all alignments that pass the filters. `-a 2` will output only reads that have a unique alignment regardless of score after applying the filters. `-a 3` will output only reads that have a unique best scoring alignment after applying the filters. If multiple alignments have the same best score, an alignment is not reported. `-a 4` will output only reads that have a best scoring alignment (possibly many best scoring alignments may exist).

3.5.2 Usage

`-r FILENAME, --rgFileName=FILENAME`

This option specifies the BFAST reference genome file (see [subsection 4.2.1](#) for the file format). This is only required if MAF or SAM output is desired (`-O 1` or `-O 3`).

`-i FILENAME, --alignedFileName=FILENAME`

Specifies the BFAST aligned file (see [subsection 4.2.5](#) for the file format).

`-a INTEGER, --algorithm=INTEGER`

This specifies the algorithm to choose the alignment for each end of the read after filtering. The option `-a 0` specifies that no filtering will occur. The option `-a 1` specifies that all alignments that pass the filters will be outputted. The option `-a 2` outputs only

reads that have been aligned uniquely. The option `-a 3` chooses uniquely the alignment with the best score. The option `-a 3` chooses all alignments with the best score.

`-O INTEGER, --outputFormat=INTEGER`

Specifies the output format. `-O 0` specifies the output to be in BFAST aligned file format (see [subsection 4.2.5](#) for the file format). `-O 1` specifies the output to be in BFAST multiple sequence alignment file format (see [subsection 4.2.9](#) for the file format). `-O 2` specifies the output to be in BFAST general feature format file format (currently undocumented and experimental). `-O 3` specifies the output to be in BFAST sequence alignment/map file format (see <https://sourceforge.net/projects/samtools/>).

3.6 butil

butil is a folder containing utilities that were developed for personal use to test, debug, and compliment the BFAST program and its accompanying publication. They are included in this distribution to aid in using BFAST and to give examples of other uses for the indexes built and data generated by BFAST. There is no support or warranty for these utilities. If options are not specified in the correct order, data may be overwritten, and crashes will certainly occur. Please use at your own risk and consult the source code if problems arise. If you find one of these utilities incredibly useful, please contact the authors/developers as to recommend a utility be supported.

Most of the utilities must have all options specified and the order listed below, without any command line flags.

3.6.1 bafconvert

`bafconvert` converts a BFAST aligned file to the specified output format.

`arg 1`

The BFAST aligned file to convert.

`arg 2`

Specifies the output type. 0 converts a binary BFAST aligned file to a text BFAST aligned file. 1 converts a text BFAST aligned file to a binary BFAST aligned file. 2 converts a binary BFAST aligned file to a BFAST multiple sequence alignment file. 3 converts a binary BFAST

aligned file to a BFAST general feature format file (currently undocumented and experimental). 4 converts a binary BFAST aligned file to a BFAST sequence alignment/map file (currently experimental, see <https://sourceforge.net/projects/samtools/>).

arg 3

Specifies the BFAST reference genome file. This option is not required for BFAST aligned file output.

arg 4

Specifies an output ID, which will be prepended to the name of each read. This option is required for BFAST sequence alignment/map file output only.

3.6.2 balignmentscoredistribution

Assess the alignment score distribution (histogram) for all reads with a given number of CALs. The alignment scores are binned according the given parameters.

arg 1

The BFAST aligned file to analyze.

arg 2

Bins from.

arg 3

Bins by.

arg 4

Bins to.

3.6.3 balignsim

balignsim generates synthetic reads given a number of variants and errors from a reference genome and tests the various local alignment algorithms.

arg 1

The BFAST reference genome file from which reads should be generated. This should have been created in nucleotide space.

arg 2

Specifies the Scoring Matrix file used to score the alignments. Please see [subsection 4.1.7](#) for the file format.

arg 3

Specifies the type of local alignment to perform. 0 specifies that full local alignment should be performed, where we consider mismatches, insertions, deletions and color errors (if `-A 1` is specified). 1 specifies that only local alignment considering mismatches should be performed and color errors (if `-A 1` is specified).

arg 4

The space in which the reads should be outputted. Use 0 for nucleotide space, and 1 for color space.

arg 5

Specifies if an indel should be added. Use 0 for no indel, 1 for a deletion, and 2 for an insertion.

arg 6

If 1 or 2 is used for arg 5, then this specifies the length of the indel.

arg 7

Specifies if errors (arg97) should be included within the indel.

arg 8

Specifies the number of SNPs to included in the read.

arg 9

Specifies the number of errors to include in the read. In nucleotide space this will result in mismatches and in color space this will result in color errors. The errors are applied after the application any indel or SNP.

arg 10

Specifies the read length to output.

arg 11

This specifies the number of reads to be generated.

arg 12

The number of threads to use for the search.

arg 13

A temporary file directory to store temporary files.

3.6.4 **berrordistribution**

`berrordistribution` counts the number of errors at each position in the read for both nucleotide space and color space, as well as counting the number of reads containing a given number of errors.

arg1

Specify 1 if we are to count insertions as errors, 0 otherwise.

arg2

Specify 1 if we are to trim the end of the alignments for a contiguous insertion, 0 otherwise. This may be desirable if the ends of reads may contain adaptor sequence.

arg3

The output ID for the output text files.

arg4

The BFAST reported files to be examined.

3.6.5 bevalsim

`bevalsim` parses a BFAST aligned file resulting from using reads generated by `bgeneratereads` to give accuracy statistics for the mapping. The output will be a text file.

arg 1

BFAST aligned file name to be evaluated.

arg 2

The reads file name generated by `bgeneratereads`.

arg 3

The output ID for the output text file.

3.6.6 bexonify

`bexonify` modifies a BFAST index file to only include locations specified by the user. The main purpose is to allow for alignment to exons, multiple subregions or the like.

arg 1

BFAST reference genome file accompanying the BFAST index file you wish to modify.

arg 2

The BFAST index file you wish to modify.

arg 3

The input exon list file name. Please consult the source code for the file format.

3.6.7 bfixbaf

`bfixbaf` updates an old BFAST aligned file to the newest format.

arg 1

Old version number.

arg 2

Specifies the Scoring Matrix file used to score the alignments. Please see [subsection 4.1.7](#) for the file format.

arg 3

Average mismatch quality.

arg 4

Specifies the encoding space of the alphabet. For nucleotide space, use `-A 0`. For color space, use `-A 1`, which will use a modified Smith Waterman algorithm to correct for color space errors.

arg 5

The input BFAST aligned file file.

3.6.8 **bfixbif**

`bfixbif` updates an old BFAST index file to the newest format.

arg 1

The input BFAST index file file.

3.6.9 **bfixbmf**

`bfixbmf` updates an old BFAST matches file to the newest format.

arg 1

The input BFAST matches file file.

3.6.10 **bfixbrg**

bfixbrg updates an old BFAST reference genome file to the newest format.

arg 1

The input BFAST reference genome file file.

3.6.11 **bfixhash**

bfixhash regenerates the hash lookup table for BFAST index file given a new hash width.

arg 1

BFAST reference genome file accompanying the BFAST index file you wish to modify.

arg 2

The BFAST index file you wish to modify.

arg 3

The new hash width that you wish to generate.

3.6.12 **bgeneratereads**

bgeneratereads generates synthetic reads given a number of variants and errors from a reference genome. See the source code for the output file format.

arg 1

The BFAST reference genome file from which reads should be generated. This should have been created in nucleotide space.

arg 2

The space in which the reads should be outputted. Use 0 for nucleotide space, and 1 for color space.

arg 3

Specifies if an indel should be added. Use 0 for no indel, 1 for a deletion, and 2 for an insertion.

arg 4

If 1 or 2 is used for arg 3, then this specifies the length of the indel.

arg 5

Specifies if errors (arg 7) should be included within the indel.

arg 6

Specifies the number of SNPs to included in the read.

arg 7

Specifies the number of errors to include in the read. In nucleotide space this will result in mismatches and in color space this will result in color errors. The errors are applied after the application any indel or SNP.

arg 8

Specifies the read length to output.

arg 9

An input of 0 specifies that the read will be single end, and an input of 1 specifies the read should be paired end. If the read is paired end, all of the variants (indel and SNPs) will be in one of the reads with the total number of errors distributed randomly across both reads.

arg 10

This specifies the length between the paired end reads (if arg 9 is specified as paired end).

arg 11

This specifies the number of reads to be generated.

3.6.13 **bheader**

`bheader` prints the header of a BFAST reference genome file or a BFAST index file.

`arg 1`

This specifies the input file. The input file can be either a BFAST reference genome file or a BFAST index file.

3.6.14 **bindexdist**

`bindexdist` prints each unique read from the genome and the number of times it occurs, where the genome is contained in the BFAST index file.

`arg 1`

The BFAST reference genome file for the accompanying BFAST index file.

`arg 2`

The BFAST index file to be examined.

`arg 3`

The number of mismatches to examine.

`arg 4`

The output directory for the output file.

`arg 5`

The output ID for the output file.

`arg 6`

A temporary file directory to store temporary files.

`arg 7`

The number of threads to use for the search.

3.6.15 bindexhist

`bindexhist` prints a histogram that counts the number of unique k -mers in the genome that occur X number of times. The k -mer chosen comes from the layout of the BFAST index file.

arg 1

The BFAST reference genome file for the accompanying BFAST index file.

arg 2

The BFAST index file to be examined.

arg 3

The number of mismatches to examine.

arg 4

The number of threads to use for the search.

3.6.16 bmergesorted

`bmergeorted` merges two sorted BFAST aligned file files.

arg 1

The first sorted BFAST aligned file.

arg 2

The second sorted BFAST aligned file.

arg 3

The output ID for the output file.

3.6.17 bmfconvert

`bmfconvert` converts a BFAST matches file from binary to text or vice versa.

arg 1

The BFAST matches file to convert.

arg 2

The input type of the BFAST matches file. 0 if the input type is text, 1 if the input type is binary.

3.6.18 **bpairedenddist**

bpairedenddist prints the distribution of the distance between paired-end reads using reads that have both ends matching only one location on the same strand from a BFAST matches file.

arg1

The output ID for the output text file.

arg 2

Specifies the minimum paired end distance to evaluate.

arg 3

Specifies the maximum paired end distance to evaluate.

arg 4

Specifies the bin size to evaluate.

arg 5

The BFAST matches files, BFAST aligned files, or BFAST reported files to examine.

3.6.19 **brepeat**

brepeat finds all contiguous repeats in the genome specified by the index that fall within the specified unit length range and minimum contiguous length.

arg 1

The BFAST reference genome file in which to search for repeats.

arg 2

The minimum unit length for a repeat.

arg 3

The maximum unit length for a repeat.

arg 4

The maximum total repeat length as a scalar multiple of the unit length.

3.6.20 **brg2fasta**

`brg2fasta` prints the reference genome in FASTA format.

arg 1

The source BFAST reference genome file.

3.6.21 **bsort**

`bsort` is a utility that sorts a BFAST reported file based on chromosome, position, strand, and alignment length.

arg 1

A BFAST reported file.

arg 2

The maximum number of entries when sorting. This should be set based on memory limitations. Please see the source code for more information.

arg 3

The number of threads to use while sorting.

`arg 4`

A temporary file directory to store temporary files.

3.6.22 `btestindexes`

`btestindexes` is a utility that tests, searches for, and compares layouts for indexes against certain events, such as errors, mismatches and insertions.

This utility can sample the space of possible indexes and the space of reads with a given set of errors and variants to find accurate index sets for use with BFAST. By specifying `-a 0`, the greedy search strategy will run. We initially seed the index set with an index with one contiguous mask. Next, we iteratively add indexes to the set as follows. We search for the best index that would increase the accuracy of the set when added. After sampling the possible space of indexes (`-s`), we choose add the best index to the set. To estimate the accuracy of an index set, we create an accuracy profile. The accuracy profile computes the accuracy for mapping reads with a specific number of SNPs/errors and color errors (see `-M` and `-E` respectively). We prioritize color errors over SNPs, meaning when comparing the accuracy profile of two index sets, we compare the accuracy for mapping reads with 1 to the specified maximum number of color errors (`-E`) with no SNPs. We repeat the comparison with one SNP, two SNPs, up to the maximum number of SNPs (`-M`).

This utility can also be used to print the accuracy for each scenario of a read with variants and errors (`-a 1`).

`-a` INTEGER

The algorithm to run. The option `-a 0` will search for masks. The option `-a 1` will compute the accuracy of masks read from file.

`-r` INT

Specifies the read length to examine.

`-S` INT

Specifies the number of events in our sampling space. This corresponds to the number of random reads to generate to estimate the accuracy for a specific scenario of events.

`-A INT`

Specifies the encoding space of the alphabet. For nucleotide space, use `-A 0`. For color space, use `-A 1`.

`-s INT`

Specifies the number of masks in our sampling space (for `-a 0`).

`-l INT`

Specifies the mask key size when sampling indexes (for `-a 0`).

`-w INT`

Specifies the maximum mask width when sampling indexes (for `-a 0`).

`-n INT`

Specifies the maximum index set size (or the maximum number of indexes in one set). Each index will be added greedily one at a time (for `-a 0`).

`-t INT`

Specifies the accuracy threshold that must be met for a specific scenario when comparing index set accuracy during sampling (for `-a 0`). Once the index set has reached this accuracy threshold for the given scenario, the next scenario will determine the index set selection.

`-f STRING`

Specifies the input file name for the masks (for `-a 1`). (See [subsection 4.1.2](#) for the file format).

`-I INT`

Specifies the maximum insertion length when evaluating index sets (for `-a 1`).

`-M INT`

Specifies the maximum number of mismatches. With `-A 0` this will correspond to SNPs or errors. With `-A 1` this will correspond to SNPs.

`-E INT`

Specifies the number of color errors to include (for `-A 1`).

`-p`

Prints the program parameters.

`-h`

Prints a help message.

3.6.23 btranslocations

Outputs unique paired end alignments for which each end is on a different contig.

`arg 1`

Specifies the input range for the first end in `contig1-contig2:pos1-pos2` (INTEGER-INTEGER:INTEGER-INTEGER) format.

`arg 2`

Specifies the input range for the second end in `contig1-contig2:pos1-pos2` (INTEGER-INTEGER:INTEGER-INTEGER) format.

`arg 3`

Specifies to accept reads for which one or both the ends fall within the specified ranges. 0 specifies that we only require one end to fall within its respective range. 1 specifies that we only require both ends to fall within its respective range.

`arg 4`

The output ID.

`arg 5`

The BFAST reported file to be examined.

3.7 scripts

scripts is a folder containing scripts that were developed to convert the input files from Illumina and ABI SOLiD sequencers to the BFAST FASTQ format. Additionally, we include a script to parallelize BFAST on a cluster. Currently only SGE and PBS clusters are supported.

3.7.1 bfast.submit.pl

This script will run BFAST on a SGE or PBS cluster. Please use the `-man` option for information on how to use the script. Note that the PERL module XML Simple is required to be installed for compilation and can be found at <http://search.cpan.org/dist/XML-Simple>.

`-help`

Print a brief help message and exits.

`-config`

Print the configuration XML schema.

`-man`

Prints the manual page and exits.

`-quiet`

Do not print any submit messages.

`-config`

The XML configuration file.

3.7.2 qseq2fastq.pl

This script will convert Illumina generated QSEQ files to the BFAST FASTQ format. Please execute the script with no arguments for more information.

3.7.3 `solid2fastq`

This program will convert ABI SOLiD generated CSFASTA and QUAL files to the BFAST format. It will also split the input reads into chunks for parallel computation. Please execute the program with no arguments for more information. We also include a PERL version of this program for developer modification.

Chapter 4

File Formats

4.1 Input Files

These files represent the input files that are used by one or more BFAST binaries but are not generated as output by a BFAST binary. Although some files are used as input to other binaries, for example the BFAST matches file is used as input to `balign`, they are described in [section 4.2](#). Examples of each input file is given in [section 4.3](#).

4.1.1 Reference genome FASTA file

The reference genome FASTA file follows the familiar FASTA format used to describe one or more molecular sequences or contigs. Each contig begins with a header line, characterized by a greater-than (>) symbol at the beginning of the line. The contig's sequence is then listed beginning on a new line. The end of the contig's sequence is specified by the end of the file or a new header line for the next contig.

An example of such a file can be seen [Figure 4.1](#). In this example, there are two contigs specified.

4.1.2 Index Layout File

The Index Layout file specifies the masks and hash width for each index you wish to generate. Each line corresponds to an index for one layout. The entries in a line should be tab or space delimited and has two entries. The first entry is the index hash width. The second entry is the mask for the index. The mask is specified as zeros and ones with no spaces. A one specifies that a base is to be used, and a zero specifies that a base is to be ignored. The

keysize is the number of ones in the mask and the width is the total number of zeros and ones.

An example can be seen in [Figure 4.2](#). In this example, the layouts for twenty indexes are specified. Each index will have a hash width of fourteen. The first index, specified on the first line, is a contiguous eighteen bases, with a keysize of eighteen and width of eighteen.

4.1.3 Exons File

This Exons file specifies an exon-like structure, with each line representing an exon. Each exon has four entries specifying the start contig, start pos, end contig, and end position in that order. An example of an Exons file can be found in [Figure 4.3](#).

4.1.4 BFAST index file file list

This type of file is simply a list of BFAST index files. Each line should correspond to one BFAST index file. An example can be seen in [Figure 4.4](#).

4.1.5 Reads FASTQ file

This file contains the reads for which we wish to align. The reads are specified in FASTQ format. The first line begins with the @ symbol. The rest of the first line will be the read name. The second line contains the sequence for the read. Currently the entire sequence must be specified one line and should be specified 5' → 3' from left-to-right. The third line will begin with the + symbol. The rest of the line can be empty or contain an arbitrary comment string. The fourth line will contain the sequence qualities.

For ABI SOLiD or color space reads, the adaptor should be included in the sequence and the colors should be encoded as [0 – 4] with 4 signifying a unknown color. There should be one Phred-like quality score for each base in the sequence (or number of colors for ABI SOLiD data).

For paired end or multi end data, each end should be specified separately but have the same read name. The should be listed in order of 5' → 3' from left-to-right and on the same strand. Multi end, paired end, or single end data can be incorporated into the same Reads FASTQ file as long as the data follows the above rules.

Another method to specify this file is through the use of a grammar:


```

<fastq>                := <fastq>@<read name><\n><info><\n>
<fastq>@<read name>    := <fastq>@<read name><\n><info><\n>@<read name>
<info>                 := <sequence><\n><comment><\n><qualities>
<read name>            := [^<\n>]+
<sequence>             := <NT sequence>
<sequence>             := <CS sequence>
<NT sequence>          := [ACGTNacgtn.]+
<CS sequence>          := [ACGT][01234.]+
<comment>              := [^<\n>]+
<qualities>            := [!~]+

```

An example of a reads file in nucleotide space can be found in [Figure 4.5](#). An example of a reads file in nucleotide space with paired end reads can be found in [Figure 4.6](#). An example of a reads file in color space can be found in [Figure 4.7](#).

4.1.6 Offsets File

The Offsets file specifies the offsets used when generating keys to find CALs. The offsets should be delimited by whitespace and in increasing order. Valid offsets include zero up to the maximum read length to be aligned. Offsets greater than the maximum read length will be ignored.

An example of an Offsets file can be found in [Figure 4.8](#). In this example odd offsets from zero to twenty are printed.

4.1.7 Scoring Matrix File

The Scoring Matrix file specifies how the local aligner should score gaps in the alignment, nucleotide substitutions, and if applicable, color substitutions.

Each entry is whitespace delimited. The first two entries represent the affine gap open penalty and the affine gap extension penalty. The next two entries represent the nucleotide substitution penalties (match then mismatch). For color space alignments, the final two entries represent the color substitution penalties (match then mismatch).

An example with a file for use with nucleotide space alignment can be found in [Figure 4.9](#). An example with a file for use with color space alignment can be found in [Figure 4.10](#).

4.2 BFAST Files

These files are generated by the BFAST utilities. Explicit examples of these files are not given since they are specified in the source code and will (hopefully) be created through the use of BFAST.

4.2.1 BFAST reference genome file

The BFAST reference genome file stores the sequence to which we wish to align. The sequence is stored in a binary format. Each base (or color) is stored in four bits: two bits for the raw base (or color), one bit to specify if the letter was an N (or a 4), and one bit to store if the base was upper case or lower case (not applicable to a color). The BFAST reference genome file stores only the forward strand. Therefore for a genome of size G (forward strand), we can estimate the total required storage size of a BFAST reference genome file to be $G/2$ bytes.

The contigs that compose the reference genome are indexed based on the order specified in the Reference genome FASTA file (see [subsection 4.1.1](#)) along with each contig's associated name (see [section 3.2](#)).

The BFAST reference genome file will have the prefix `bfast.rg.file` and the file extension `.brg`. Information about the BFAST reference genome file can be found by using the binary `bheader` (see [subsection 3.6.13](#)). Please see the source code for the full internal binary representation.

4.2.2 BFAST index file

The BFAST index file stores the index and hash table for the BFAST reference genome file. The index and hash table are stored in a binary format, with only the forward strand indexed.

To estimate the required storage size of an index before creation, we must know the number of contigs, hash width and genome size (forward strand). It is interesting to note that the BFAST reference genome file size does neither depend on the keysize, key width, nor mask layout. If there are more than 256 contigs in the BFAST reference genome file then each starting position indexed will require 8 bytes of storage. If there are 256 or fewer contigs in the BFAST reference genome file then each starting position indexes will require 5 bytes of storage. This representation is handled internally and is not visible to the user. Since we index a four letter alphabet, the hash with width j will require 4×4^j bytes (4 bytes per hash entry). Thus if the genome size is G (forward strand), the estimated BFAST index file required storage size is approximately $5 \times G + 4 \times 4^j$ or $8 \times G + 4 \times 4^j$ for a small number (≤ 256) or large number (> 256) contigs respectively.

The BFAST index file will have the prefix `bfast.index.file` and the file extension `.bif`. Information about the BFAST index file can be found by using the binary `bheader` (see [subsection 3.6.13](#)). Please see the source code for the full internal binary representation.

4.2.3 BFAST matches file

The BFAST matches file is used to store Candidate Alignment Locations (CALs) for each read processed by `bmatches` (see [section 3.3](#)). By default, this file is stored in binary format. This file can be converted to text format for manual inspection by using the utility `bmfconvert` (see [subsection 3.6.17](#)).

To estimate the file size *a priori* is difficult. The read length, read name length, and number of CALs for each read must be known. The factor that causes the majority of the file size bloat is the average number of CALs stored per read. This can be overcome by having an upper limit on the number of CALs to store (see [section 3.3](#)).

The BFAST matches file will have the prefix `bfast.matches.file` and file extension `.bmf`. The file format for the text version of the BFAST matches file is as follows.

All entries are tab delimited. The first line has two entries: the `@` symbol appended to the read name, and number of ends of the read. The number of subsequent lines corresponds to the number of ends in the read. For each end of the read, we have the original reads sequence, original quality values, a flag indicated whether the maximum CALs was reached, the number of CALs found (0 if the maximum was reached), and the CALs. Each CAL has three fields: the contig (1-based), position (1-based), and strand. Please see the source code for the full internal binary representation.

4.2.4 BFAST reads filtered file

The BFAST reads filtered file has the same format as the reads FASTQ file (see [subsection 4.1.5](#)) and is used to store reads that were initially filtered by `bmatches` (see [section 3.3](#)).

The BFAST reads filtered file will have the prefix `bfast.reads.filtered.file` and file extension `.fa`.

4.2.5 BFAST aligned file

The BFAST aligned file is used to store the alignments of reads to the reference genome and is created by the utility `balign` (see [section 3.4](#)). By default, this file is stored in binary format. This file can be converted to text format for manual inspection by using the utility `bafconvert` (see [subsection 3.6.1](#)).

To estimate the file size *a priori* is difficult. The read length, read name length, and number of alignments, and the length of the alignments for each read must be known. The factor that causes the majority of the file size bloat is the average number of alignments stored per read. This can be overcome by having an upper limit on the number of alignments per read to consider (see [section 3.4](#)) or by filtering the alignments (see [section 3.5](#)).

The BFAST aligned file will have the prefix `bfast.aligned.file` and the file extension `.baf`.

The first line contains three fields: the @ symbol prepended to the read name, the encoding of the read (0 for NT space and 1 for color space), and the number of ends in the read. For each end of the read, the first line contains three fields: the original sequence, the original quality values, and the number of alignments. For each alignment for each end, there are three (NT space) or four (color space) lines. The first line contains eight fields: the contig name, the contig number (1-based), the position (1-based), the strand, the alignment score, the mapping quality, the reference length (the length of the alignment minus the sum of the lengths of any insertions), and the length of the total alignment including insertions and deletions. The second line contains the reference sequence. The third line contains the read sequence. For color space, there is a fourth line, which contains the called color error information.

Please see the source code for the full internal binary representation.

4.2.6 BFAST not aligned file

The BFAST not aligned file is created by `balign` (see [section 3.4](#)) to store reads that had no alignments or were flagged for having too many Candidate Alignment Locations (CALs). The underlying file format is a BFAST matches file (see [subsection 4.2.3](#)).

The BFAST not aligned file will have the prefix `bfast.not.aligned.file` and the file extension `.bnaf`.

4.2.7 BFAST reported file

The BFAST reported file is a file storing the alignments chosen by `bpostprocess` (see [section 3.5](#)). The underlying format is selected as an option in `bpostprocess` and can be either a BFAST aligned file or a BFAST multiple sequence alignment file.

The BFAST reported file will have the prefix `bfast.report.file` and the file extension according to the underlying format.

4.2.8 BFAST not reported file

The BFAST not reported file is a file that stores those reads for which alignments could not be chosen by `bpostprocess` (see [section 3.5](#)). The underlying format is selected as an option in `bpostprocess` and can be either a BFAST aligned file or a BFAST multiple sequence alignment file.

The BFAST not reported file will have the prefix `bfast.not.reported.file` and the file extension according to the underlying format.

4.2.9 BFAST multiple sequence alignment file

The BFAST multiple sequence alignment file is a file format to store multiple sequence alignments. This format conforms to the UCSC specification of multiple alignment format (see <http://genome.ucsc.edu/FAQ/FAQformat#format5>). The file is stored in text format.

For a detailed description of the multiple alignment format see <http://genome.ucsc.edu/FAQ/FAQformat#format5>). Each entry represents one alignment for a read. Depending on the options used to generate this file (see [section 3.5](#)), there may be multiple entries for a read. Paired end reads have each end outputted individually. Alignment score, paired end information including to which end the alignment belong, contig name and optionally color error information is printed on the “a” line. The next two “s” lines follow the defined format conventions.

4.3 Example Input Files

```

>NM_006435 2
gaggaaactgttgagaaaacggaactactggggaaagggaggggtcactg
agaaccatcccggtaacccgatcacctggtcaccatgaaccacattgt
gcaaaccttctctcctgtcaacagcggccagcctcccaactacgagatgc
tcaaggaggagcaggaagtggctatgctgggggtgccccacaacctgct
ccccgatgtccaccgtgatccacatccgcagcgagacctccgtgcctga
ccatgtggtctggtccctgttcaacacctcttcatgaacacctgctgcc
tgggcttcatagcattcgcgactccgtgaagtctagggaacaggaagatg
gttggcgacgtgaccggggccaggcctatgcctccaccgccaagtgcct
gaacatctgggccctgattttgggcaccttcatgaccattctgctcatca
tcatcccagtggttggtcgtccaggcccagcgatagatcaggaggcatcat
tgaggccaggagctctgcccgtgacctgtatcccacgtactctatcttcc
attcctcgccctgccccagaggccaggagctctgcccttgacctgtatt
ccacttactccaccttccattcctcgccctgtccccacagccgagtcctg
catcagccctttatcctcacacgcttttctacaatggcattcaataaagt
gtatatgtttctggtgctgctgtgacttcaaaaaaaaaa
>NM_015644 3
agtgctctcttccgccttcagtgccctgctcatcaagggtctgggtttcc
cggtcctctggcgaggatcctccaaggcgtctcacatgaaccgggtcaga
aacgccaaaatctacgtggagagagctgtcaagaagaagatctttacaat
ccaaggctgctaccgggtgatccgggtgtctcttgcgccggaggggctggg
tgagagaagaagatggtccatcgctcaggccccacctgcgcc
tgggcttcatagcattcgcgactccgtgaagtctagggaacaggaagatg
gttggcgacgtgaccggggccaggcctatgcctccaccgccaagtgcct
gaacatctgggccctgattttgggcaccttcatgaccattctgctcatca
tcatcccagtggttggtcgtccaggcccagcgatagatcaggaggcatcat

```

Figure 4.1: An example of a reference genome FASTA file.
See [subsection 4.1.1](#) for a description.

```

14 111111111111111111
14 111111111011111111
14 1111101111110101111
14 1111101100110011101111
14 111101110101111101111
14 1110110111101101100101111
14 11111001001111001111111
14 1111011111111111111
14 11111010110101101111111
14 11101111111001110111011
14 11101101111111011010111
14 1111101011101111101111
14 1111011100101111001011111
14 1110101111001001111110111
14 111101111111111000101111
14 11111000111111111110011
14 11111111011100001010110111
14 1110110111011101001110111
14 1110110110110110111001111
14 1110110001100011101111111

```

Figure 4.2: An example of an Index Layout file.
See [subsection 4.1.2](#) for a description.

```

1 891540 1 892246
1 895320 1 896847
1 897118 1 897867
1 897904 1 900541
1 1129098 1 1129929
1 1130413 1 1130935
1 1131428 1 1132152
1 1256389 1 1259906
1 2312874 1 2313457
1 2316883 1 2317370

```

Figure 4.3: An example of an Exons file
See [subsection 4.1.3](#) for a description.

```

/indexes/nt.space/hg18/bfast.index.file.hg18.0.1.1.25.16571.0.1.bif
/indexes/nt.space/hg18/bfast.index.file.hg18.0.1.1.25.16571.0.2.bif
/indexes/nt.space/hg18/bfast.index.file.hg18.0.1.1.25.16571.0.3.bif
/indexes/nt.space/hg18/bfast.index.file.hg18.0.1.1.25.16571.0.4.bif
/indexes/nt.space/hg18/bfast.index.file.hg18.0.1.1.25.16571.0.5.bif
/indexes/nt.space/hg18/bfast.index.file.hg18.0.1.1.25.16571.0.6.bif
/indexes/nt.space/hg18/bfast.index.file.hg18.0.1.1.25.16571.0.7.bif
/indexes/nt.space/hg18/bfast.index.file.hg18.0.1.1.25.16571.0.8.bif
/indexes/nt.space/hg18/bfast.index.file.hg18.0.1.1.25.16571.0.9.bif
/indexes/nt.space/hg18/bfast.index.file.hg18.0.1.1.25.16571.0.10.bif

```

Figure 4.4: An example of a BFAST index file list file.
See [subsection 4.1.4](#) for a description.

```

@4:150:844:843
GAGCGTATCGAGGCTCTAAAAAGATGTATACTAGCATTCTTCTCT
+
IIIIIIII*III3IIIIIIIIIIIIIIIIII,?II<1III+IIIII
@4:150:353:142
TGATTCATATCATGATGCTGGTAAACATTTTCTTTATGGTTCTCT
+
II-II.IIIIIIE*%&II%&II%II?4II/8I%9I.(I((2%&6%B
@4:150:495:390
TTCGCATGTTTCTCCTTTTTTTTCCCCTTCTTTCCTCTTCTTTT
+
III4?IIIIIIDIIIIIIIIII3IIIIII8II7%, '2&?I%*-)II

```

Figure 4.5: An example of a reads FASTQ file in nucleotide space.
See [subsection 4.1.5](#) for a description.


```

@4:150:276:201
TTATGCTAATTTGCATACTGACCAAGAACGTGATTACTTCATTCA
+
IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII3BII&)
@4:150:276:201
TTGTATGTTTTTCATGCCTCCAAATCTTTGAGGCTTTTTTTTTTTTT
+
IIIIIIIIIIII8IIIIIIIIII1II1II*>I+=/IIIIII;IIIII
@4:150:495:344
TGATTATGACCAGTGTTTCCAGTCCGTTTTTTTTTTTTTTTTTTCT
+
IIIIIIIIIIIIIIIIIIIIII5IIIIICI?$III?I6I1%III#;
@4:150:495:344
TCTCACGTTGGCTGACGACCGCTTTGTGGCGTTTTTTTTATATTCT
+
IIIIIIIIIIIIIIII3IIIIIB%<I1B) 7I' IFE+I+I' 'C (%<&%

```

Figure 4.6: An example of a reads FASTQ file in nucleotide space with paired end reads.
See [subsection 4.1.5](#) for a description.

```

@5_20_383
T11232310330012102133010223110131101021211013311230
+
><912>4679?6,.-)/*/40=&/3=:&4309,.168'1.-...',6&*)
@5_20_1125
T12001200003103013013302121123331111300002333112310
+
B?>7?@:8129?..+685/93>2.+6>60,(,<,&&&%&'&*,&' /2' & (
@5_20_1365
T30320113323301030133032013330330333013323033332313
+
0<?;?50+6:67%562925:9?,129?$+1-1$7./+1%5)&-3(',&%
@5_21_71
T10310020310300313122223311321123130022031131111010
+
@0/@@4<.6;/ '2>@??978,-' '12.+1.&+%&'+'%+*0*$/%%)$)&

```

Figure 4.7: An example of a reads FASTQ file in color space.
See [subsection 4.1.5](#) for a description.

1	3	5	7	9	11	13	15	17	19
---	---	---	---	---	----	----	----	----	----

Figure 4.8: An example of an Offsets file.
See [subsection 4.1.6](#) for a description.

-175
-50
50
-150

Figure 4.9: An example of a Scoring Matrix file for nucleotide space.
See [subsection 4.1.7](#) for a description.

-175
-50
50
-150
0
-125

Figure 4.10: An example of a Scoring Matrix file for color space.
See [subsection 4.1.7](#) for a description.

Chapter 5

Advanced Topics

In this chapter we present a few different applications of BFAST, as well as how to design indexes.

5.1 How To Design Indexes

This section is meant as a brief introduction on how to design indexes. It will outline what you must ask to define indexes for your specific experiment.

We assume that you have a specific reference genome to which you wish to align, reads with a known length, an alphabet (ex. A,C,G, and T) with a known size, an intuitive feeling of what error-rate and polymorphism-rate you wish to tolerate, and the amount of time you wish to wait for BFAST to complete.

We first begin by determining the optimal key-size that corresponds to your reference genome size, alphabet size, and key length. As you may have read from the BFAST paper (?), we wish to make the lookup in an index return on average one CAL. Given a genome of size G , a key size of k , reads of length L , and an alphabet of size A , we compute the expected number of “false” random key matches F to be

$$F = (L - k + 1) \times \frac{G}{A^k}$$

This can be expressed in R code as:

```
# Calculates the number of ``false`` random key matches
# in the index given:
#   L: read length
#   k: key size
```

```
# G: genome size (both + and -)
# A: alphabet (DNA is 4)
F <- function(L=50, k=18, G=2*3.2*10^9, A=4)
{
  return ( (L - k + 1) * (G / (A^k)) );
}
```

In this case, we test k over a varying range of values to find the smallest value of k where F is less than one. In theory this is the key size we should use. In practice, for larger genomes, the distribution of bases is non-random, for example in the Human Genome there are many stretches of long repeats. Therefore we advise you to choose a key size of $k + 2$, to further guarantee the uniqueness of the lookup to be performed. For the Human Genome, key sizes of 18 or greater will suffice.

After deciding on our key size, we move to explicitly creating the masks for the indexes. This is achieved by using the binary utility `btestindexes` (see [subsection 3.6.22](#)). This utility must be run twice, first to find a set of masks, and a second time to estimate the accuracy of those masks. We suggest using a key width (number of zeros and ones in your mask) greater than your key size (required) but also smaller than your read length, since the smaller the key width the more offsets can be used during the lookup step(?).

Nevertheless, using the `btestindexes` utility allows the user to examine various mask sets and their associated estimated accuracy against many possible error and variant combinations. We recommend that the user selects the minimal number of masks sufficient to tolerate the user's desired accuracy tolerances. The fewer masks used, the faster the alignment will be performed.

After finding a set of masks to use for alignment, the final step is to select the hash width to use. The hash accelerates the lookup by building an index of the prefixes of all possible keys in the index. In general, the hash width will take an exponential amount of space relative to the given hash width. For example, a good hash width for the Human Genome is 14, which will add approximately 1GB to the index size (see [subsection 4.2.2](#)). For smaller genomes, much smaller hash widths can be used.

Finally, both disk storage and random access memory sizes need to be considered. Based on all of the parameters above, it is easy to calculate the required size in bytes of each index to be created. This can be found in [subsection 4.2.2](#).

If the indexes become too large, we urge you to upgrade your machine with more random access memory and disk space given cost of such an upgrade compared to the actual generation (sequencing) of the data. If all attempts at convincing the President fail, we suggest you further divide the indexes by ranges across the genome. This can be achieved by using the `-s`, `-S`, `-e`, and `-E` in `bpreprocess` (see [section 3.2](#)).

5.2 Whole-Genome Alignment

Whole-Genome alignment is as simple as following the work flow presented in [section 2.2](#). An example can be found in [section 6.1](#).

5.3 Targeted Genomic Alignments

There are a number of ways to target specific regions within the genome, for example by specifying a subset of chromosomes or a number of contiguous regions.

The first method is to use `bpreprocess` and to specify an Exons file (see [section 3.2](#)). The second method is to use `bexonify` to create new targeted indexes from already created indexes (see [subsection 3.6.6](#)). The third method is to use command line options to limit the starting contig and position, and ending contig and position.

Applications of this type of index creation included targeted pull-down methods, where it is known only a certain set of regions will be sequenced.

5.3.1 Using `bpreprocess` and exon list

To target specific regions of a larger reference genome, we can specify a Exons file when creating the indexes in `bpreprocess` using the `-x` option (see [section 3.2](#)). This will limit the locations indexed to just those specified in the Exons file. Subsequently, options that limit the number of CALs returned by a key look up or in total for a read (see [section 3.3](#) and [section 3.4](#)) will only be relative to this reduced index.

5.3.2 Using `bexonify` on existing indexes

If BFAST index files of a reference genome have already been created, then `bexonify` (see [subsection 3.6.6](#)) can filter the BFAST index files as to only include the desired targeted regions. This can save time by avoiding having to create new indexes.

5.3.3 Using command-line options to specify one contiguous range

The command line options `-s`, `-S`, `-e`, and `-E` can be used to only consider one contiguous range within the BFAST reference genome file. This specified during the index creation (see [section 3.2](#)), during the local alignment step (see [section 3.4](#)), or when prioritizing alignments (see [section 3.5](#)). The step at which these options are specified will affect the resulting output.

If specified during the index creation step, the BFAST index file will only contain the sequence from that region. Thus, only CALs within this range will be found. If we are

limiting the number of CALs returned by a key or in total for a read (see [section 3.3](#) and [section 3.4](#)), then only CALs within the range will count towards these limits.

If specified during the local alignment step, only CALs that fall within the specified range will produce alignments. If we are limiting the number of CALs returned by a key or in total for a read (see [section 3.3](#) and [section 3.4](#)), then all CALs that are possible in the indexes will count towards these limits. For example, if the index is of the whole genome, but we are interested in one chromosome, then if the CAL limits are used the limits are imposed according to CALs found in the whole genome, not the specified region. This may be useful if we want to flag reads that have high homology to a larger region or genome, but to have the alignments only be outputted within a specified range. Furthermore, since the local alignment step is typically the most expensive step for computation, ignoring alignments outside a certain range will reduce the number of local alignments needed.

If specified when prioritizing alignments, only alignments within the specified range will be outputted. This is similar to limiting the alignment range using `balign` but is useful when a BFAST aligned file has been created with alignments to the full reference genome and we wish to only report alignments within a contiguous region.

5.4 Transcriptome Alignment

In some cases a contiguous reference genome is not the desired reference sequence. Examples include alignment to the transcriptome, including different transcript of genes, splice variants, or isoforms. This type of alignment can be easily handled by BFAST. We refer to each transcript, splice variant, isoform, or contiguous sequence as a contig.

The sequence each possible transcripts should be given as independent contig in the Reference genome FASTA file when creating the BFAST reference genome file (see [section 3.2](#)). This will ensure that each transcript will be indexed separately and reported separately. Each step of the work flow (see [section 2.2](#)) should proceed as normal.

The contigs will be given an index number based on the order specified in the Reference genome FASTA file as well as outputting their name as defined in the Reference genome FASTA file. In this manner the ID of the contig can be recovered. In a BFAST matches file (see [subsection 4.2.3](#)) generated by `bmatches` (see [section 3.3](#)) only the index number is given for compactness. In a BFAST aligned file (see [subsection 4.2.5](#)) generated by `balign` (see [section 3.4](#)) either the index number or original contig name can be used. In a BFAST multiple sequence alignment file (see [subsection 4.2.9](#)) generated by `bpostprocess` (see [section 3.5](#)) either the index number or original contig name can be used.

5.5 Bisulfite Treated or Methylation Alignment

Bisulfite sequencing is an interesting experiment whereby we wish to know the methylation status of certain bases. We assume that in the sequence data some of the C bases have been converted to T by bisulfite treatment. In this case, we wish to align the sequence data to a reference genome, tolerating the fact that a fair number of mismatches when aligned will come from the fact that Cs have been converted to Ts.

BFAST can support this type of alignment. In brief, we will “methylate” the reference genome, methylate the sequence reads to be aligned (and annotate where those conversions were made), align the converted reads to the reference genome, then finally convert the reads back to their original state using the annotations.

Suppose we have 25 contigs representing the 25 chromosomes of the Human Genome. We convert each strand of each chromosome by changing every C to a T, for a total of 50 final methylated contigs (this must be done independently by the user). We use this “converted” reference genome as input when creating a BFAST reference genome file.

Next we convert every C to a T in each read in our input Reads FASTQ file (this must be done independently by the user). We can either annotate where each conversion occurred, or just store the original read. Either way, the annotation or the original read can be appended to the read name, since this will be kept throughout by BFAST.

After converting the reference sequence and the input reads, we run BFAST using the standard work flow (see [section 2.2](#)) with two exceptions. The first exception is that in `bmatches` we wish to use the option `-w 1` so that we only match to the forward strand of each contig (see [section 3.3](#)). The reason for this is that we will index each strand separately. The second exception is that we have reduced our alphabet size from four (A, C, G, and T) to three (A, G, and T). The reduced alphabet size must be taken into consideration when deciding on the masks for our indexes, since our genome complexity has been reduced. This should lead the user to use a larger key size to combat this reduced complexity.

After alignment using BFAST is complete, we simply convert back the read to its original state (this must be done independently by the user) thereby giving us the locations where there are Ts in the reference and Cs (or Ts) in the read.

5.6 Color Space Alignment

The work flow for color space has five steps as seen in [Figure 5.1](#) similar in fashion to the one described in [section 2.2](#).

1. In the first step we build two reference genomes: a nucleotide space genome using the option `-A 0` in `bpreprocess`, and a color space genome using the option `-A 1` (see

[section 3.2](#)).

2. In the second step we create the indexes in color space by using the color space reference genome built in the first step and the option `-A 1` (see [section 3.2](#)).
3. In the third step we search for CALs using the color space indexes created in the second step, using the color space reference genome built in first step, and by using the option `-A 1` (see [section 3.3](#)).
4. In the fourth step we perform local alignment using the nucleotide space reference genome built in the first step and by using the option `-A 1` (see [section 3.4](#)).
5. In the fifth step we prioritize the local alignments as was previously described in [section 2.2](#).

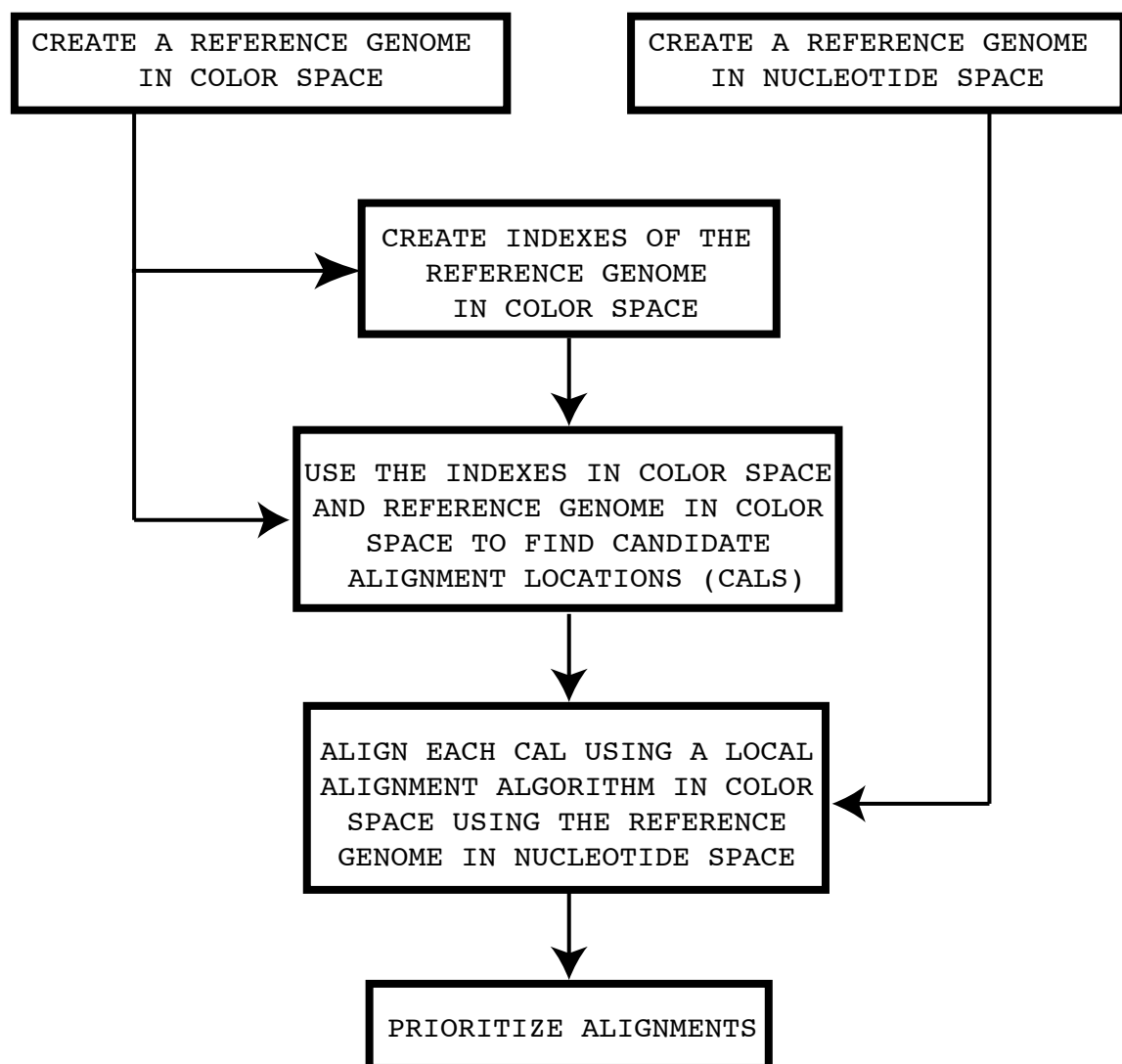


Figure 5.1: The BFAST work flow for color space alignment.
See [section 5.6](#) for a description.

Chapter 6

Examples

6.1 Whole-Genome Alignment of Phi X 174

In this example, we will align read to the Phi X 174 genome using the work flow presented in [section 2.2](#).

To begin this example, open the URL: <http://genome.ucla.edu/bfast/examples>. Right-click the file “bfast-example-phi.x.174.tar.gz” and select the “save-as” option to download the examples file. To uncompress the compressed archive, in the command line type in “tar zxvf bfast-example-phi.x.174.tar.gz” (no quotes).

We first want to create a BFAST reference genome file (see [section 3.2](#)). The reference genome FASTA file is stored in a file called “phi.x.174.fa”. The command we use to create the BFAST reference genome file is:

```
$bpreprocess -r phi.x.174.fa -a 0 -o phi.x.174 -d ./
```

This should create a file called “bfast.rg.file.phi.x.174.0.brg”, which is the BFAST reference genome file.

The second step is to create the indexes using the BFAST reference genome file as input (see [section 3.2](#)). We must also specify layouts for our indexes. We give five layouts in “layouts.txt” that are chosen for 50 base-pair reads, and robust against mismatches. The method used to choose these indexes is not described here, but can be performed using `btestindexes` (see [subsection 3.6.22](#) and [section 5.1](#)). The command we use to create the BFAST index files is:

```
$bpreprocess -r bfast.rg.file.phi.x.174.0.brg -i layouts.txt -a 1 -o phi.x.174 -d ./ -T ./
```

This should create five files, one for each BFAST index file:

```
bfast.index.file.phi.x.174.0.1.1.1.5386.0.1.bif
bfast.index.file.phi.x.174.0.1.1.1.5386.0.2.bif
bfast.index.file.phi.x.174.0.1.1.1.5386.0.3.bif
bfast.index.file.phi.x.174.0.1.1.1.5386.0.4.bif
bfast.index.file.phi.x.174.0.1.1.1.5386.0.5.bif
```

The third step is to find candidate alignment locations (CALs) for the set of reads using the indexes created in the second step (see [section 3.3](#)). We have generated synthetic reads from the Phi X 174 genome using `bgeneratereads` (see [subsection 3.6.12](#)). We created reads with 0 to 5 mismatches, with 10,000 reads created for each scenario, for a total of 60,000 reads. The reads file is “reads.phi.x.174.fa”. In this example we use all five indexes as our main indexes (listed in “main.indexes.txt”), and no secondary indexes (see “secondary.indexes.txt”). We use all possible offsets, listed in “offsets.txt”. The command we use to find the matches is:

```
$bmatches -r bfast.rg.file.phi.x.174.0.brg -i main.indexes.txt -I secondary.indexes.txt -R reads.phi.x.174.fa
-O offsets.txt -o phi.x.174 -d ./ -T ./
```

Two files should be outputted, a BFAST matches file (“bfast.matches.file.phi.x.174.bmf”) and a BFAST reads filtered file (“bfast.reads.filtered.file.phi.x.174.0.0.fa”).

The fourth step is to run the local alignment algorithm for each CAL for a given read (see [section 3.4](#)). We use the scoring matrix found in “scoring.matrix.txt” and the BFAST matches file created in the previous step. The command we use to perform the local alignment is:

```
$balign -r bfast.rg.file.phi.x.174.0.brg -m bfast.matches.file.phi.x.174.bmf -x scoring.matrix.txt -o phi.x.174
-d ./ -T ./
```

This should create two output files, a BFAST aligned file (“bfast.aligned.file.phi.x.174.baf”) and a BFAST not aligned file (“bfast.not.aligned.file.phi.x.174.bmf”).

The fifth and final step is to prioritize the alignments (see [section 3.5](#)). We use as input the BFAST aligned file created in the previous step. In this case, we choose to output the best unambiguous alignment for each read. The command we use to prioritize alignments is:

```
$bpostprocess -i bfast.aligned.file.phi.x.174.baf -a 3 -o phi.x.174 -d ./ -O 0
```

This should create two output files, a BFAST reported file (“bfast.reported.file.phi.x.174.baf”) and a BFAST not reported file (“bfast.not.reported.file.phi.x.174.baf”). Since the BFAST reported file will be in binary format, to view the alignments in text we use the utility `bafconvert` (see [subsection 3.6.1](#)):

```
$bafconvert bfast.reported.file.phi.x.174.baf 1
```

This should create a file called “bfast.reported.file.phi.x.174.text.baf”, which will be the BFAST reported file but in text format.

Chapter 7

Appendix

7.1 Human Genome Alignment Recommended Settings

We assume that you have the full hg18 reference in the FASTA format in a file called hg18.fa. We detail our recommended commands to align sensitively to the human genome, at the cost of speed, outputting in the SAM format (see <http://samtools.sourceforge.net>). Further filtering, especially filtering on mapping quality, alignment score, and alignment quality, should be also be performed. For a more detailed explanation, please see [section 6.1](#). Please use the `-n` option for multi-threaded alignment where possible. For Illumina data see [subsection 7.1.1](#). For ABI SOLiD data see [subsection 7.1.2](#)

7.1.1 Illumina

We assume your reads are in Illumina QSEQ format (the files that end with `_qseq.txt`) and we wish to align lane *X*. Note, we align one lane all at one time but splitting the converted reads file allows for parallelism.

The `secondary.indexes.txt` file should exist but be empty. The `scoring.matrix.txt` can be found at [Figure 4.9](#). For reads less than > 40bp, the `main.indexes.txt` should be:

```

14 11111111111111111111
14 111010001110001110100011011111
14 11110100110111101010101111
14 11111111111111001111
14 11110111011001010011111111
14 11110111000101010000010101110111
14 1011001101011110100110010010111
14 1110110010100001000101100111001111
14 11110111111111111111
14 11011111100010110111101101

```

For reads greater than or equal to 40bp, the `main.indexes.txt` should be:

```

14 11111111111111111111
14 1111101110111010100101011011111
14 1011110101101001011000011010001111111
14 10111001101001100100111101010001011111
14 111110110111011110111111111
14 111111100101001000101111101110111
14 11110101110010100010101101010111111
14 111101101011011001100000101101001011101
14 1111011010001000110101100101100110100111
14 1111010010110110101110010110111011

```

Given the above files, the commands we should execute are:

Convert the reads:

```
$perl bfast-0.5.7/scripts/qseq2fastq.pl s_X > reads.s_X.fastq
```

Convert the reference:

```
$bfast-0.5.7/bpreprocess/bpreprocess -r hg18.fa -a 0 -o hg18 -d ./
```

Create the indexes:

```
$bfast-0.5.7/bpreprocess/bpreprocess -r bfast.rg.file.hg18.0.brg -i layouts.txt -a 1 -o hg18 -d ./
-T ./
```

Search the indexes:

```
$bfast-0.5.7/bmatches/bmatches -r bfast.rg.file.hg18.0.brg -i main.indexes.txt -I secondary.indexes.txt
-R reads.s_X.fastq -O offsets.txt -K 8 -M 384 -o hg18 -d ./ -T ./
```

Perform local alignment:

```
$bfast-0.5.7/balign/balign -r bfast.rg.file.hg18.0.brg -m bfast.matches.file.hg18.bmf -x scoring.matrix.txt
-M 384 -o hg18 -d ./ -T ./
```


Filter alignments:

```
$bfast-0.5.7/bpostprocess/bpostprocess -i bfast.aligned.file.hg18.baf -r bfast.rg.file.hg18.0.brg -a
3 -o hg18 -d ./ -O 3
```

7.1.2 ABI SOLiD

We assume your reads are at least 50bp in length. We will split the input into 10,000,000 read pieces for parallel computation.

The `secondary.indexes.txt` file should exist but be empty. The `scoring.matrix.txt` can be found at [Figure 4.10](#). The `main.indexes.txt` should be:

```
14 11111111111111111111111111111111
14 11111010011111100111111111111111
14 1011111101100110001111110001111111
14 1111111100101111000001100011111011
14 11111111000111111001111111111111
14 111110110100110000110001100111111111
14 11111111111110011101111111111111
14 11101100001111111100111101111111
14 1110110001011010011100101111101111
14 111111001000110001011100110001100011111
```

Given the above files, the commands we should execute are:

Convert the reads:

```
$perl bfast-0.5.7/scripts/solid2fastq.pl -n 10000000 -o reads *.csfasta *.qual
```

Convert the reference (nucleotide space):

```
$bfast-0.5.7/bpreprocess/bpreprocess -r hg18.fa -a 0 -o hg18 -d ./
```

Convert the reference (color space):

```
$bfast-0.5.7/bpreprocess/bpreprocess -r hg18.fa -a 0 -A 1 -o hg18 -d ./
```

Create the indexes:

```
$bfast-0.5.7/bpreprocess/bpreprocess -r bfast.rg.file.hg18.1.brg -i layouts.txt -a 1 -A 1 -o hg18 -d
./ -T ./
```

Search the indexes:

```
$bfast-0.5.7/bmatches/bmatches -r bfast.rg.file.hg18.1.brg -i main.indexes.txt -I secondary.indexes.txt
-R reads.n.fastq -O offsets.txt -A 1 -K 8 -M 384 -o hg18.n -d ./ -T ./
```

Perform local alignment:

```
$bfast-0.5.7/balign/balign -r bfast.rg.file.hg18.0.brg -m bfast.matches.file.hg18.n.bmf -x scoring.matrix.txt  
-A 1 -M 384 -o hg18 -d ./ -T ./
```

Filter alignments:

```
$bfast-0.5.7/bpostprocess/bpostprocess -i bfast.aligned.file.hg18.n.baf -r bfast.rg.file.hg18.0.brg  
-a 3 -o hg18 -d ./ -O 3
```

Note that for parallel computation, execute `bmatches`, `balign`, and `bpostprocess` for each converted input file created (replace *n* with the input file number).

7.2 High-Speed Tutorial

Not for the faint of heart, most details will be omitted.

Your best bet is to follow the work flow in [section 2.2](#) (or [section 5.6](#) for color space). A quick list of relevant sections are as follows:

- Step 1: [subsection 3.2.1](#).
- Step 2: [subsection 3.2.2](#).
- Step 3: [subsection 3.3.1](#).
- Step 4: [subsection 3.4.1](#).
- Step 5: [subsection 3.5.1](#).

7.3 Copyright

GNU GENERAL PUBLIC LICENSE Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation’s software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author’s protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors’ reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone’s free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections

when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly

provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program

under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

one line to give the program's name and a brief idea of what it does. Copyright (C)
 year name of author

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) year name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the

GNU Lesser General Public License instead of this License.