khmer Documentation

Release 1.1-197-g38c1358-dirty

2010-2014 Michael R. Crusoe, Greg Edvenson, Jordan Fish, Adina

August 03, 2014

Contents

Authors Michael R. Crusoe, Greg Edvenson, Jordan Fish, Adina Howe, Luiz Irber, Eric McDonald, Joshua Nahum, Kaben Nanlohy, Humberto Ortiz-Zuazaga, Jason Pell, Jared Simpson, Camille Scott, Ramakrishnan Rajaram Srinivasan, Qingpeng Zhang, and C. Titus Brown

Contact khmer-project@idyll.org

khmer is a library and suite of command line tools for working with DNA sequence. It is primarily aimed at short-read sequencing data such as that produced by the Illumina platform. khmer takes a k-mer-centric approach to sequence analysis, hence the name.

Installing and running khmer

There are two mailing lists dedicated to khmer, an announcements-only list and a discussion list. To search their archives and sign-up for them, please visit the following URLs:

- Discussion: http://lists.idyll.org/listinfo/khmer
- Announcements: http://lists.idyll.org/listinfo/khmer-announce

The archives for the khmer list are available at: http://lists.idyll.org/pipermail/khmer/

khmer development has largely been supported by AFRI Competitive Grant no. 2010-65205-20361 from the USDA NIFA, and is now funded by the National Human Genome Research Institute of the National Institutes of Health under Award Number R01HG007513, both to C. Titus Brown.

Contents:

CHAPTER 1

Introduction to khmer

1.1 Introduction

khmer is a library and toolkit for doing k-mer-based dataset analysis and transformations. Our focus in developing it has been on scaling assembly of metagenomes and mRNA.

khmer can be used for a number of transformations, include inexact transformations (abundance filtering and error trimming) and exact transformations (graph-size filtering, to throw away disconnected reads; and partitioning, to split reads into disjoint sets). Of these, only partitioning is not constant memory. In all cases, the memory required for assembly with Velvet or another de Bruijn graph assembler will be more than the memory required to use our software. Our software will not increase the memory required for Velvet, either, although we may not be able to *decrease* the memory required for assembly for every data set.

Most of khmer relies on an underlying probabilistic data structure known as a Bloom filter (also see MinCount Sketch), which is essentially a set of hash tables, each of different size, with no collision detection. These hash tables are used to store the presence of specific k-mers and/or their count. The lack of collision detection means that the Bloom filter may report a k-mer as being "present" when it is not, in fact, in the data set; however, it will never incorrectly report a k-mer as being absent when it *is* present. This one-sided error makes the Bloom filter very useful for certain kinds of operations.

khmer is also independent of K, and currently works for K <= 32. We will be integrating code for up to K=64 soon.

khmer is implemented in C++ with a Python wrapper, which is what all of the scripts use.

1.2 Using khmer

khmer comes "out of the box" with a number of scripts that make it immediately useful for a few different operations, including:

- normalizing read coverage ("digital normalization")
- dividing reads into disjoint sets that do not connect ("partitioning")
- eliminating reads that will not be used by a de Bruijn graph assembler;
- removing reads with low- or high-abundance k-mers;
- trimming reads of certain kinds of sequencing errors;
- counting k-mers and estimating data set coverage based on k-mer counts;
- running Velvet and calculating assembly statistics;
- optimizing assemblies on various parameters;

- converting FASTA to FASTQ;

and a few other random functions.

1.3 Practical considerations

The most important thing to think about when using khmer is whether or not the transformation or filter you're applying is appropriate for the data you're trying to assemble. Two of the most powerful operations available in khmer, graph-size filtering and graph partitioning, only make sense for assembly datasets with many theoretically unconnected components. This is typical of metagenomic data sets.

The second most important consideration is memory usage. The effectiveness of all of the Bloom filter-based functions (which is everything interesting in khmer!) depends critically on having enough memory to do a good job. See *Choosing table sizes for khmer* for more information.

1.4 Copyright and license

CHAPTER 2

Contributors and Acknowledgements

khmer is a product of the GED lab at Michigan State University,

http://ged.msu.edu/

—

C. Titus Brown <ctb@msu.edu> wrote the initial ktable and hashtable implementations, as well as hashbits and counting_hash.

Jason Pell implemented many of the C++ k-mer filtering functions.

Qingpeng contributed code to do unique k-mer counting.

Adina Howe, Rosangela Canino-Koning, and Arend Hintze contributed significantly to discussions of approaches and algorithms; Adina wrote a number of scripts.

Jared T. Simpson (University of Cambridge, Sanger Institute) contributed paired-end support for digital normalization.

Eric McDonald thoroughly revised many aspects of the code base, made much of the codebase thread safe, and otherwise improved performance dramatically.

Michael R. Crusoe is the new maintainer of khmer.

MRC 2014-05-07

CHAPTER 3

The khmer user documentation

Contents:

3.1 Installing and running khmer

You'll need Python 2.7+ and internet access.

The khmer project currently works with Python 2.6 but we target Python 2.7+.

3.1.1 Build requirements

OS X

1. From a terminal download the virtualenv package and create a virtual environment with it:

```
curl -O https://pypi.python.org/packages/source/v/virtualenv/virtualenv-1.11.6.tar.gz
tar xzf virtualenv*
cd virtualenv-*; python2.7 virtualenv.py ../khmerEnv; cd ..
source khmerEnv/bin/activate
```

Linux

1. Install the python development environment, virtualenv, pip, gcc, and g++.
   - On recent Debian and Ubuntu this can be done with:

     ```
     sudo apt-get install python2.7-dev python-virtualenv python-pip gcc \
           g++
     ```

   - For RHEL6:

     ```
     sudo yum install -y python-devel python-pip git gcc gcc-c++ make
     sudo pip install virtualenv
     ```

2. Create a virtualenv and activate it:

   ```
   cd a/writable/directory/
   python2.7 -m virtualenv khmerEnv
   source khmerEnv/bin/activate
   ```

   Linux users without root access can try the OS X instructions above.

3.1.2 Installing khmer inside the virtualenv

1. Use pip to download, build, and install khmer and its dependencies:

   ```
   pip2 install khmer
   ```

2. The scripts are now in the `env/bin` directory and ready for your use. You can directly use them by name, see *khmer's command-line interface*.

3. When returning to khmer after installing it you will need to reactivate the virtualenv first:

   ```
   source khmerEnv/bin/activate
   ```

Run the tests

After installing you can run the embedded test suite:

```
nosetests khmer --attr '!known_failing'
```

If the nosetests binary isn't installed then:

```
pip2 install khmer[tests]
nosetests khmer --attr '!known_failing'
```

3.2 A few examples

See the 'examples' subdirectory for complete examples.

3.2.1 STAMPS data set

The 'stamps' data set is a fake metagenome-like data set containing two species, mixed at a 10:1 ratio. The source genomes are in 'data/stamps-genomes.fa'. The reads file is in 'data/stamps-reads.fa.gz', and consists of 100-base reads with a 1% error rate.

The example shows how to construct k-mer abundance histograms, as well as the effect of digital normalization and partitioning on the k-mer abundance distribution.

See the script for running everything and the IPython Notebook.

For an overall discussion and some slides to explain what's going on, visit the Web site for a 2013 HMP metagenome assembly webinar that Titus Brown gave.

3.3 An assembly handbook for khmer - rough draft

date 2012-11-2

An increasing number of people are asking about using our assembly approaches for things that we haven't yet written (or posted) papers about. Moreover, our assembly strategies themselves are also under constant evolution as we do more research and find ever-wider applicability of our approaches.

Note, this is an exact copy of Titus' blog post, here – go check the bottom of that for comments.

3.3.1 Authors

This handbook distills the cumulative expertise of Adina Howe, Titus Brown, Erich Schwarz, Jason Pell, Camille Scott, Elijah Lowe, Kanchan Pavangadkar, Likit Preeyanon, and others.

3.3.2 Introduction

khmer is a general framework for low-memory k-mer counting, filtering, and advanced trickery.

The latest source is always available here.

khmer is really focused on short read data, and, more specifically, Illumina, because that's where we have a too-much-data problem. However, a lot of the prescriptions below can be adapted to longer read technologies such as 454 and Ion Torrent without much effort.

Don't try to use our k-mer approaches with PacBio – the error rate is too high.

There are currently two papers available on khmer: the partitioning paper and the digital normalization paper.

There are many blog posts about this stuff on Titus Brown's blog. We will try to link them in where appropriate.

3.3.3 Asking for help

There's some documentation here:

> https://khmer.readthedocs.org/en/latest/

There's also a khmer mailing list at lists.idyll.org that you can use to get help with khmer. To sign up, just go to the khmer lists page and subscribe.

3.3.4 Preparing your sequences

Do all the quality filtering, trimming, etc. that you think you should do.

Most of the khmer tools currently work "out of the box" on interleaved paired-end data. Ask on the list if you're not sure.

All of our scripts will take in .fq or .fastq files as FASTQ, and all other files as FASTA. gzip files are always accepted. Let us know if not; that's a bug!

Most scripts *output* FASTA, and some mangle headers. Sorry. We're working on outputting FASTQ for FASTQ input, and removing any header mangling.

3.3.5 Picking k-mer table sizes and k parameters

For k-mer table sizes, read *Choosing table sizes for khmer*

For k-mer sizes, we recommend k=20 for digital normalization and k=32 for partitioning; then assemble with a variety of k parameters.

3.3.6 Genome assembly, including MDA samples and highly polymorphic genomes

1. Apply digital normalization as follows.

Broadly, normalize each insert library separately, in the following way:

For high-coverage libraries (> ~50x), do three-pass digital normalization: run normalize-by-median to C=20 and then run filter-abund with C=1. Now split out the remaining paired-end/interleaved and single-end reads using strip-and-split-for-assembly, and normalize-by-median the paired-end and single-end files to C=5 (in that order).

For low-coverage libraries (< 50x) do single-pass digital normalization: run normalize-by-median to C=10.

2. Extract any remaining paired-end reads and lump remaining orphan reads into singletons using strip-and-split-for-assembly

3. Then assemble as normal, with appropriate insert size specs etc. for the paired end reads.

You can read about this process in the digital normalization paper.

3.3.7 mRNAseq assembly

1. Apply single-pass digital normalization.

Run normalize-by-median to C=20.

2. Extract any remaining paired-end reads and lump remaining orphan reads into singletons using strip-and-split-for-assembly

3. Then assemble as normal, with appropriate insert size specs etc. for the paired end reads.

You can read about this process in the digital normalization paper.

3.3.8 Metagenome assembly

1. Apply single-pass digital normalization.

Run normalize-by-median to C=20 (we've also found C=10 works fine).

2. Run filter-below-abund with C=50 (if you diginormed to C=10) or C=100 (if you diginormed to C=20);

3. Partition reads with load-graph, etc. etc.

4. Assemble groups as normal, extracting paired-end reads and lumping remaining orphan reads into singletons using strip-and-split-for-assembly.

(We actually use Velvet at this point, but there should be no harm in using a metagenome assembler such as MetaVelvet or MetaIDBA or SOAPdenovo.)

Read more about this in the partitioning paper. We have some upcoming papers on partitioning and metagenome assembly, too; we'll link those in when we can.

3.3.9 Metatranscriptome assembly

(Not tested by us!)

1. Apply single-pass digital normalization.

Run normalize-by-median to C=20.

2. Extract any remaining paired-end reads and lump remaining orphan reads into singletons using strip-and-split-for-assembly

3. Then assemble with a genome or metagenome assembler, *not* an mRNAseq assembler. Use appropriate insert size specs etc. for the paired end reads.

3.3.10 Preprocessing Illumina for other applications

(Not tested by us!)

Others have told us that you can apply digital normalization to Illumina data prior to using Illumina for RNA scaffolding or error correcting PacBio reads.

Our suggestion for this, based on no evidence whatsoever, is to diginorm the Illumina data to C=20.

3.3.11 Quantifying mRNAseq or metagenomes assembled with digital normalization

For now, khmer only deals with assembly! So: assemble. Then, go back to your original, unnormalized reads, and map those to your assembly with e.g. bowtie. Then count as you normally would :).

3.3.12 Philosophy of digital normalization

The basic philosophy of digital normalization is "load your most valuable reads first." Diginorm gets rid of redundancy iteratively, so you are more likely to retain the first reads fed in; this means you should load in paired end reads, or longer reads, first.

3.3.13 Iterative and independent normalization

You can use *--loadtable* and *--savetable* to do iterative normalizations on multiple files in multiple steps. For example, break

```
normalize-by-median.py [ ... ] file1.fa file2.fa file3.fa
```

into multiple steps like so:

```
normalize-by-median.py [ ... ] --savetable file1.kh file1.fa
normalize-by-median.py [ ... ] --loadtable file1.kh --savetable file2.kh file2.fa
normalize-by-median.py [ ... ] --loadtable file2.kh --savetable file3.kh file3.fa
```

The results should be identical!

If you want to independently normalize multiple files for speed reasons, go ahead. Just remember to do a combined normalization at the end. For example, instead of

```
normalize-by-median.py [ ... ] file1.fa file2.fa file3.fa
```

you could do

```
normalize-by-median.py [ ... ] file1.fa
normalize-by-median.py [ ... ] file2.fa
normalize-by-median.py [ ... ] file3.fa
```

and then do a final

```
normalize-by-median.py [ ... ] file1.fa.keep file2.fa.keep file3.fa.keep
```

The results will not be identical, but should not differ significantly. The multipass approach will take more total time but may end up being faster walltime because you can execute the independent normalizations on multiple computers.

For a cleverer approach that we will someday implement, read the Beachcomber's Dilemma.

3.3.14 Validating and comparing assemblies

More here soon :).

3.4 khmer's command-line interface

The simplest way to use khmer's functionality is through the command line scripts, located in the scripts/ directory of the khmer distribution. Below is our documentation for these scripts. Note that all scripts can be given `-h` which will print out a list of arguments taken by that script.

Many scripts take `-x` and `-N` parameters, which drive khmer's memory usage. These parameters depend on details of your data set; for more information on how to choose them, see *Choosing table sizes for khmer*.

You can also override the default values of `--ksize`/`-k`, `--n_tables`/`-N`, and `--min-tablesize`/`-x` with the environment variables *KHMER_KSIZE*, *KHMER_N_TABLES*, and *KHMER_MIN_TABLESIZE* respectively.

1. *k-mer counting and abundance filtering*
2. *Partitioning*
3. *Digital normalization*
4. *Read handling: interleaving, splitting, etc.*

Note: Almost all scripts take in either FASTA and FASTQ format, and output the same. Some scripts may only recognize FASTQ if the file ending is '.fq' or '.fastq', at least for now.

Files ending with '.gz' will be treated as gzipped files, and files ending with '.bz2' will be treated as bzip2'd files.

3.4.1 k-mer counting and abundance filtering

load-into-counting.py

Build a k-mer counting table from the given sequences.

usage: load-into-counting.py [-h] [–version] [-q] [–ksize KSIZE] [–n_tables N_TABLES] [–min-tablesize MIN_TABLESIZE] [–threads N_THREADS] [-b] [–report-total-kmers] output_countingtable_filename input_sequence_filename [input_sequence_filename ...]

output_countingtable_filename
The name of the file to write the k-mer counting table to.

input_sequence_filename
The names of one or more FAST[AQ] input sequence files.

-h, --help
show this help message and exit

--version
show program's version number and exit

-q, --quiet

--ksize <int>, **-k** <int>
k-mer size to use

--n_tables <int>, **-N** <int>
number of k-mer counting tables to use

--min-tablesize <float>, **-x** <float>
lower bound on tablesize to use

--threads <int>, **-T** <int>
Number of simultaneous threads to execute

-b, --no-bigcount
Do not count k-mers past 255

--report-total-kmers, -t
Prints the total number of k-mers to stderr

Note: with *-b* the output will be the exact size of the k-mer counting table and this script will use a constant amount of memory. In exchange k-mer counts will stop at 255. The memory usage of this script with *-b* will be about 1.15x the product of the *-x* and *-N* numbers.

Example:

```
load-into-counting.py -k 20 -x 5e7 out.kh data/100k-filtered.fa
```

Multiple threads can be used to accelerate the process, if you have extra cores to spare.

Example:

```
load_into_counting.py -k 20 -x 5e7 -T 4 out.kh data/100k-filtered.fa
```

abundance-dist.py

Calculate abundance distribution of the k-mers in the sequence file using a pre-made k-mer counting table.

usage: abundance-dist.py [-h] [-z] [-s] [–version] input_counting_table_filename input_sequence_filename output_histogram_filename

input_counting_table_filename
The name of the input k-mer counting table file.

input_sequence_filename
The name of the input FAST[AQ] sequence file.

output_histogram_filename
The columns are: (1) k-mer abundance, (2) k-mer count, (3) cumulative count, (4) fraction of total distinct k-mers.

-h, --help
show this help message and exit

-z, --no-zero
Do not output 0-count bins

-s, --squash
Overwrite output file if it exists

--version
show program's version number and exit

abundance-dist-single.py

Calculate the abundance distribution of k-mers from a single sequence file.

usage: abundance-dist-single.py [-h] [–version] [-q] [–ksize KSIZE] [–n_tables N_TABLES] [–min-tablesize MIN_TABLESIZE] [–threads THREADS] [-z] [-b] [-s] [–savetable filename] [–report-total-kmers] input_sequence_filename output_histogram_filename

input_sequence_filename
The name of the input FAST[AQ] sequence file.

output_histogram_filename
The name of the output histogram file. The columns are: (1) k-mer abundance, (2) k-mer count, (3) cumulative count, (4) fraction of total distinct k-mers.

-h, --help
show this help message and exit

--version
show program's version number and exit

-q, --quiet

--ksize <int>, **-k** <int>
k-mer size to use

--n_tables <int>, **-N** <int>
number of k-mer counting tables to use

--min-tablesize <float>, **-x** <float>
lower bound on tablesize to use

--threads <int>, **-T** <int>
Number of simultaneous threads to execute

-z, --no-zero
Do not output 0-count bins

-b, --no-bigcount
Do not count k-mers past 255

-s, --squash
Overwrite output file if it exists

--savetable <filename>
Save the k-mer counting table to the specified filename.

--report-total-kmers, -t
Prints the total number of k-mers to stderr

Note that with *-b* this script is constant memory; in exchange, k-mer counts will stop at 255. The memory usage of this script with *-b* will be about 1.15x the product of the *-x* and *-N* numbers.

To count k-mers in multiple files use **load_into_counting.py** and **abundance_dist.py**.

filter-abund.py

Trim sequences at a minimum k-mer abundance.

usage: filter-abund.py [-h] [–threads THREADS] [–cutoff CUTOFF] [–variable-coverage] [–normalize-to NORMALIZE_TO] [-o optional_output_filename] [–version] input_presence_table_filename input_sequence_filename [input_sequence_filename ...]

input_presence_table_filename
The input k-mer presence table filename

input_sequence_filename
Input FAST[AQ] sequence filename

-h, --help
show this help message and exit

--threads <int>, **-T** <int>
Number of simultaneous threads to execute

--cutoff <int>, **-C** <int>
Trim at k-mers below this abundance.

--variable-coverage, -V
Only trim low-abundance k-mers from sequences that have high coverage.

--normalize-to <int>, **-Z** <int>
Base the variable-coverage cutoff on this median k-mer abundance.

-o <optional_output_filename>, **--out** <optional_output_filename>
Output the trimmed sequences into a single file with the given filename instead of creating a new file for each input file.

--version
show program's version number and exit

Trimmed sequences will be placed in ${input_sequence_filename}.abundfilt for each input sequence file. If the input sequences are from RNAseq or metagenome sequencing then *--variable-coverage* should be used.

Example:

```
load-into-counting.py -k 20 -x 5e7 table.kh data/100k-filtered.fa
filter-abund.py -C 2 table.kh data/100k-filtered.fa
```

filter-abund-single.py

Trims sequences at a minimum k-mer abundance (in memory version).

usage: filter-abund-single.py [-h] [–version] [-q] [–ksize KSIZE] [–n_tables N_TABLES] [–min-tablesize MIN_TABLESIZE] [–threads THREADS] [–cutoff CUTOFF] [–savetable filename] [–report-total-kmers] input_sequence_filename

input_sequence_filename
FAST[AQ] sequence file to trim

-h, --help
show this help message and exit

--version
show program's version number and exit

-q, --quiet

--ksize <int>, **-k** <int>
k-mer size to use

--n_tables <int>, **-N** <int>
number of k-mer counting tables to use

--min-tablesize <float>, **-x** <float>
lower bound on tablesize to use

--threads <int>, **-T** <int>
Number of simultaneous threads to execute

--cutoff <int>, **-C** <int>
Trim at k-mers below this abundance.

--savetable <filename>
If present, the name of the file to save the k-mer counting table to

--report-total-kmers, -t
Prints the total number of k-mers to stderr

Trimmed sequences will be placed in ${input_sequence_filename}.abundfilt.

This script is constant memory.

To trim reads based on k-mer abundance across multiple files, use **load-into-counting.py** and **filter-abund.py**.

Example:

```
filter-abund-single.py -k 20 -x 5e7 -C 2 data/100k-filtered.fa
```

count-median.py

Count k-mers summary stats for sequences

usage: count-median.py [-h] [–version] input_counting_table_filename input_sequence_filename output_summary_filename

input_counting_table_filename
input k-mer count table filename

input_sequence_filename
input FAST[AQ] sequence filename

output_summary_filename
output summary filename

-h, --help
show this help message and exit

--version
show program's version number and exit

Count the median/avg k-mer abundance for each sequence in the input file, based on the k-mer counts in the given k-mer counting table. Can be used to estimate expression levels (mRNAseq) or coverage (genomic/metagenomic).

The output file contains sequence id, median, average, stddev, and seq length.

NOTE: All 'N's in the input sequences are converted to 'G's.

count-overlap.py

Count the overlap k-mers which are the k-mers appearing in two sequence datasets.

usage: count-overlap.py [-h] [–version] [-q] [–ksize KSIZE] [–n_tables N_TABLES] [–min-tablesize MIN_TABLESIZE] input_presence_table_filename input_sequence_filename output_report_filename

input_presence_table_filename
input k-mer presence table filename

input_sequence_filename
input sequence filename

output_report_filename
output report filename

-h, --help
show this help message and exit

--version
show program's version number and exit

-q, --quiet

--ksize <int>, **-k** <int>
k-mer size to use

--n_tables <int>, **-N** <int>
number of k-mer counting tables to use

--min-tablesize <float>, **-x** <float>
lower bound on tablesize to use

An additional report will be written to ${output_report_filename}.curve containing the increase of overlap k-mers as the number of sequences in the second database increases.

3.4.2 Partitioning

do-partition.py

Load, partition, and annotate FAST[AQ] sequences

usage: do-partition.py [-h] [–version] [-q] [–ksize KSIZE] [–n_tables N_TABLES] [–min-tablesize MIN_TABLESIZE] [–subset-size SUBSET_SIZE] [–no-big-traverse] [–threads N_THREADS] [–keep-subsets] graphbase input_sequence_filename [input_sequence_filename ...]

graphbase
base name for output files

input_sequence_filename
input FAST[AQ] sequence filenames

-h, --help
show this help message and exit

--version
show program's version number and exit

-q, --quiet

--ksize <int>, **-k** <int>
k-mer size to use

--n_tables <int>, **-N** <int>
number of k-mer counting tables to use

--min-tablesize <float>, **-x** <float>
lower bound on tablesize to use

--subset-size <float>, **-s** <float>
Set subset size (usually 1e5-1e6 is good)

--no-big-traverse
Truncate graph joins at big traversals

--threads, -T
Number of simultaneous threads to execute

--keep-subsets
Keep individual subsets (default: False)

Load in a set of sequences, partition them, merge the partitions, and annotate the original sequences files with the partition information.

This script combines the functionality of **load-graph.py**, **partition-graph.py**, **merge-partitions.py**, and **annotate-partitions.py** into one script. This is convenient but should probably not be used for large data sets, because **do-partition.py** doesn't provide save/resume functionality.

load-graph.py

Load sequences into the compressible graph format plus optional tagset.

usage: load-graph.py [-h] [–version] [-q] [–ksize KSIZE] [–n_tables N_TABLES] [–min-tablesize MIN_TABLESIZE] [–threads N_THREADS] [–no-build-tagset] [–report-total-kmers] output_presence_table_filename input_sequence_filename [input_sequence_filename ...]

output_presence_table_filename
output k-mer presence table filename.

input_sequence_filename
input FAST[AQ] sequence filename

-h, --help
show this help message and exit

--version
show program's version number and exit

-q, --quiet

--ksize <int>, **-k** <int>
k-mer size to use

--n_tables <int>, **-N** <int>
number of k-mer counting tables to use

--min-tablesize <float>, **-x** <float>
lower bound on tablesize to use

--threads <int>, **-T** <int>
Number of simultaneous threads to execute

--no-build-tagset, -n
Do NOT construct tagset while loading sequences

--report-total-kmers, -t
Prints the total number of k-mers to stderr

See **extract-partitions.py** for a complete workflow.

partition-graph.py

Partition a sequence graph based upon waypoint connectivity

usage: partition-graph.py [-h] [–stoptags filename] [–subset-size SUBSET_SIZE] [–no-big-traverse] [–version] [–threads THREADS] basename

basename
basename of the input k-mer presence table + tagset files

-h, --help
show this help message and exit

--stoptags <filename>, **-S** <filename>
Use stoptags in this file during partitioning

--subset-size <float>, **-s** <float>
Set subset size (usually 1e5-1e6 is good)

--no-big-traverse
Truncate graph joins at big traversals

--version
show program's version number and exit

--threads <int>, **-T** <int>
Number of simultaneous threads to execute

The resulting partition maps are saved as '${basename}.subset.#.pmap' files.

See 'Artifact removal' to understand the stoptags argument.

merge-partition.py

Merge partition map '.pmap' files.

usage: merge-partition.py [-h] [–ksize KSIZE] [–keep-subsets] [–version] graphbase

graphbase
basename for input and output files

-h, --help
show this help message and exit

--ksize <int>, **-k** <int>
k-mer size (default: 32)

--keep-subsets
Keep individual subsets (default: False)

--version
show program's version number and exit

Take the ${graphbase}.subset.#.pmap files and merge them all into a single ${graphbase}.pmap.merged file for **annotate-partitions.py** to use.

annotate-partitions.py

Annotate sequences with partition IDs.

usage: annotate-partitions.py [-h] [–ksize KSIZE] [–version] graphbase input_sequence_filename [input_sequence_filename ...]

graphbase
basename for input and output files

input_sequence_filename
input FAST[AQ] sequences to annotate.

-h, --help
show this help message and exit

--ksize <int>, **-k** <int>
k-mer size (default: 32)

--version
show program's version number and exit

Load in a partitionmap (generally produced by partition-graph.py or merge-partitions.py) and annotate the sequences in the given files with their partition IDs. Use **extract-partitions.py** to extract sequences into separate group files.

Example (results will be in `random-20-a.fa.part`):

```
load-graph.py -k 20 example tests/test-data/random-20-a.fa
partition-graph.py example
merge-partitions.py -k 20 example
annotate-partitions.py -k 20 example tests/test-data/random-20-a.fa
```

extract-partitions.py

Separate sequences that are annotated with partitions into grouped files.

usage: extract-partitions.py [-h] [–max-size MAX_SIZE] [–min-partition-size MIN_PART_SIZE] [–no-output-groups] [–output-unassigned] [–version] output_filename_prefix input_partition_filename [input_partition_filename ...]

output_filename_prefix

input_partition_filename

-h, --help
show this help message and exit

--max-size <int>, **-X** <int>
Max group size (n sequences)

--min-partition-size <int>, **-m** <int>
Minimum partition size worth keeping

--no-output-groups, -n
Do not actually output groups files.

--output-unassigned, -U
Output unassigned sequences, too

--version
show program's version number and exit

Example (results will be in `example.group0000.fa`):

```
load-graph.py -k 20 example tests/test-data/random-20-a.fa
partition-graph.py example
merge-partitions.py -k 20 example
annotate-partitions.py -k 20 example tests/test-data/random-20-a.fa
extract-partitions.py example random-20-a.fa.part
```

```
(extract-partitions.py will produce a partition size distribution
in <base>.dist. The columns are: (1) number of reads, (2) count
of partitions with n reads, (3) cumulative sum of partitions,
(4) cumulative sum of reads.)
```

Artifact removal

The following scripts are specialized scripts for finding and removing highly-connected k-mers (HCKs). See *Partitioning large data sets (50m+ reads)*.

make-initial-stoptags.py

Find an initial set of highly connected k-mers.

usage: make-initial-stoptags.py [-h] [–version] [-q] [–ksize KSIZE] [–n_tables N_TABLES] [–min-tablesize MIN_TABLESIZE] [–subset-size SUBSET_SIZE] [–stoptags filename] graphbase

graphbase
basename for input and output filenames

-h, --help
show this help message and exit

--version
show program's version number and exit

-q, --quiet

--ksize <int>, **-k** <int>
k-mer size to use

--n_tables <int>, **-N** <int>
number of k-mer counting tables to use

--min-tablesize <float>, **-x** <float>
lower bound on tablesize to use

--subset-size <float>, **-s** <float>
Set subset size (default 1e4 is prob ok)

--stoptags <filename>, **-S** <filename>
Use stoptags in this file during partitioning

Loads a k-mer presence table/tagset pair created by load-graph.py, and does a small set of traversals from graph waypoints; on these traversals, looks for k-mers that are repeatedly traversed in high-density regions of the graph, i.e. are highly connected. Outputs those k-mers as an initial set of stoptags, which can be fed into partition-graph.py, find-knots.py, and filter-stoptags.py.

The k-mer counting table size options parameters are for a k-mer counting table to keep track of repeatedly-traversed k-mers. The subset size option specifies the number of waypoints from which to traverse; for highly connected data sets, the default (1000) is probably ok.

find-knots.py

Find all highly connected k-mers.

usage: find-knots.py [-h] [–n_tables N_TABLES] [–min-tablesize MIN_TABLESIZE] [–version] graphbase

graphbase
Basename for the input and output files.

-h, --help
show this help message and exit

--n_tables <int>, **-N** <int>
number of k-mer counting tables to use

--min-tablesize <float>, **-x** <float>
lower bound on the size of the k-mer counting table(s)

--version
show program's version number and exit

Load an k-mer presence table/tagset pair created by load-graph, and a set of pmap files created by partition-graph. Go through each pmap file, select the largest partition in each, and do the same kind of traversal as in **make-initial-stoptags.py** from each of the waypoints in that partition; this should identify all of the HCKs in that partition. These HCKs are output to <graphbase>.stoptags after each pmap file.

Parameter choice is reasonably important. See the pipeline in *Partitioning large data sets (50m+ reads)* for an example run.

This script is not very scalable and may blow up memory and die horribly. You should be able to use the intermediate stoptags to restart the process, and if you eliminate the already-processed pmap files, you can continue where you left off.

filter-stoptags.py

Trim sequences at stoptags.

usage: filter-stoptags.py [-h] [–ksize KSIZE] [–version] input_stoptags_filename input_sequence_filename [input_sequence_filename ...]

input_stoptags_filename

input_sequence_filename

-h, --help
show this help message and exit

--ksize <int>, **-k** <int>
k-mer size

--version
show program's version number and exit

Load stoptags in from the given .stoptags file and use them to trim or remove the sequences in <file1-N>. Trimmed sequences will be placed in <fileN>.stopfilt.

3.4.3 Digital normalization

normalize-by-median.py

Do digital normalization (remove mostly redundant sequences)

usage: normalize-by-median.py [-h] [–version] [-q] [–ksize KSIZE] [–n_tables N_TABLES] [–min-tablesize MIN_TABLESIZE] [-C CUTOFF] [-p] [-s filename] [-R filename] [-f] [–save-on-failure] [-d DUMP_FREQUENCY] [-o filename] [–report-total-kmers] [-l filename] input_sequence_filename [input_sequence_filename ...]

input_sequence_filename
Input FAST[AQ] sequence filename.

-h, --help
show this help message and exit

--version
show program's version number and exit

-q, --quiet

--ksize <int>, **-k** <int>
k-mer size to use

--n_tables <int>, **-N** <int>
number of k-mer counting tables to use

--min-tablesize <float>, **-x** <float>
lower bound on tablesize to use

-C <int>, **--cutoff** <int>

-p, --paired

-s <filename>, **--savetable** <filename>

-R <filename>, **--report** <filename>

-f, --fault-tolerant
continue on next file if read errors are encountered

--save-on-failure
Save k-mer counting table when an error occurs

-d <int>, **--dump-frequency** <int>
dump k-mer counting table every d files

-o <filename>, **--out** <filename>
only output a single file with the specified filename

--report-total-kmers, -t
Prints the total number of k-mers post-normalization to stderr

-l <filename>, **--loadtable** <filename>
load a precomputed k-mer table from disk

Discard sequences based on whether or not their median k-mer abundance lies above a specified cutoff. Kept sequences will be placed in <fileN>.keep.

Paired end reads will be considered together if *-p* is set. If either read will be kept, then both will be kept. This should result in keeping (or discarding) each sequencing fragment. This helps with retention of repeats, especially.

With *-s*/*--savetable*, the k-mer counting table will be saved to the specified file after all sequences have been processed. With *-d*, the k-mer counting table will be saved every d files for multifile runs; if *-s* is set, the specified name will be used, and if not, the name *backup.ct* will be used. *-l*/*--loadtable* will load the specified k-mer counting table before processing the specified files. Note that these tables are are in the same format as those produced by **load-into-counting.py** and consumed by **abundance-dist.py**.

-f/*--fault-tolerant* will force the program to continue upon encountering a formatting error in a sequence file; the k-mer counting table up to that point will be dumped, and processing will continue on the next file.

Example:

```
normalize-by-median.py -k 17 tests/test-data/test-abund-read-2.fa
```

Example:

```
normalize-by-median.py -p -k 17 tests/test-data/test-abund-read-paired.fa
```

Example:

```
normalize-by-median.py -k 17 -f tests/test-data/test-error-reads.fq tests/test-data/test-fastq-reads
```

Example:

```
normalize-by-median.py -k 17 -d 2 -s test.ct tests/test-data/test-abund-read-2.fa tests/test-data/tes
```

3.4.4 Read handling: interleaving, splitting, etc.

extract-long-sequences.py

Extract FASTQ or FASTA sequences longer than specified length (default: 200 bp).

usage: extract-long-sequences.py [-h] [-o OUTPUT] [-l LENGTH] input_filenames [input_filenames ...]

input_filenames
: Input FAST[AQ] sequence filename.

-h, --help
: show this help message and exit

-o, --output
: The name of the output sequence file.

-l <int>**, --length** <int>
: The minimum length of the sequence file.

extract-paired-reads.py

Take a mixture of reads and split into pairs and orphans.

usage: extract-paired-reads.py [-h] [–version] infile

infile

-h, --help
: show this help message and exit

--version
: show program's version number and exit

The output is two files, <input file>.pe and <input file>.se, placed in the current directory. The .pe file contains interleaved and properly paired sequences, while the .se file contains orphan sequences.

Many assemblers (e.g. Velvet) require that you give them either perfectly interleaved files, or files containing only single reads. This script takes files that were originally interleaved but where reads may have been orphaned via error filtering, application of abundance filtering, digital normalization in non-paired mode, or partitioning.

Example:

```
extract-paired-reads.py tests/test-data/paired.fq
```

fastq-to-fasta.py

Converts FASTQ format (.fq) files to FASTA format (.fa).

usage: fastq-to-fasta.py [-h] [-o OUTPUT] [-n] input_sequence

input_sequence
The name of the input FASTQ sequence file.

-h, --help
show this help message and exit

-o, --output
The name of the output FASTA sequence file.

-n, --n_keep
Option to drop reads containing 'N's in input_sequence file.

interleave-reads.py

Produce interleaved files from R1/R2 paired files

usage: interleave-reads.py [-h] [-o filename] [–version] infiles [infiles ...]

infiles

-h, --help
show this help message and exit

-o <filename>, **--output** <filename>

--version
show program's version number and exit

The output is an interleaved set of reads, with each read in <R1> paired with a read in <R2>. By default, the output goes to stdout unless *-o*/*--output* is specified.

As a "bonus", this file ensures that read names are formatted in a consistent way, such that they look like the pre-1.8 Casava format (@name/1, @name/2).

Example:

```
interleave-reads.py tests/test-data/paired.fq.1 tests/test-data/paired.fq.2 -o paired.fq
```

sample-reads-randomly.py

Uniformly subsample sequences from a collection of files

usage: sample-reads-randomly.py [-h] [-N NUM_READS] [-M MAX_READS] [-S NUM_SAMPLES] [-R RANDOM_SEED] [-o output_file] [–version] filenames [filenames ...]

filenames

-h, --help
show this help message and exit

-N <int>, **--num_reads** <int>

-M <int>, **--max_reads** <int>

-S <int>, **--samples** <int>

-R <int>, **--random-seed** <int>

-o <output_file>, **--output** <output_file>

--version
show program's version number and exit

Take a list of files containing sequences, and subsample 100,000 sequences (*-N*/*--num_reads*) uniformly, using reservoir sampling. Stop after first 100m sequences (*-M*/*--max_reads*). By default take one subsample, but take *-S*/*--samples* samples if specified.

The output is placed in *-o*/*--output* <file> (for a single sample) or in <file>.subset.0 to <file>.subset.S-1 (for more than one sample).

This script uses the reservoir sampling algorithm.

split-paired-reads.py

Split interleaved reads into two files, left and right.

usage: split-paired-reads.py [-h] [–version] infile

infile

-h, --help
show this help message and exit

--version
show program's version number and exit

Some programs want paired-end read input in the One True Format, which is interleaved; other programs want input in the Insanely Bad Format, with left- and right- reads separated. This reformats the former to the latter.

Example:

```
split-paired-reads.py tests/test-data/paired.fq
```

3.5 Blog posts and additional documentation

3.5.1 Hashtable and filtering

The basic inexact-matching approach used by the hashtable code is described in this blog post:

http://ivory.idyll.org/blog/jul-10/kmer-filtering

A test data set (soil metagenomics, 88m reads, 10gb) is here:

http://angus.ged.msu.edu.s3.amazonaws.com/88m-reads.fa.gz

3.5.2 Illumina read abundance profiles

khmer can be used to look at systematic variations in k-mer statistics across Illumina reads; see, for example, this blog post:

http://ivory.idyll.org/blog/jul-10/illumina-read-phenomenology

The fasta-to-abundance-hist and abundance-hist-by-position scripts can be used to generate the k-mer abundance profile data, after loading all the k-mer counts into a .kh file:

```
# first, load all the k-mer counts:
load-into-counting.py -k 20 -x 1e7 25k.kh data/25k.fq.gz

# then, build the '.freq' file that contains all of the counts by position
python sandbox/fasta-to-abundance-hist.py 25k.kh data/25k.fq.gz

# sum across positions.
python sandbox/abundance-hist-by-position.py data/25k.fq.gz.freq > out.dist
```

The hashtable method 'dump_kmers_by_abundance' can be used to dump high abundance k-mers, but we don't have a script handy to do that yet.

You can assess high/low abundance k-mer distributions with the hi-lo-abundance-by-position script:

```
load-into-counting.py -k 20 25k.kh data/25k.fq.gz
python sandbox/hi-lo-abundance-by-position.py 25k.kh data/25k.fq.gz
```

This will produce two output files, <filename>.pos.abund=1 and <filename>.pos.abund=255.

3.6 Choosing table sizes for khmer

If you look at the documentation for the scripts (*khmer's command-line interface*) you'll see two mysterious parameters – `-N` and `-x`, or, more verbosely, `-n_tables` and `--tablesize`. What are these, and how do you specify them?

3.6.1 The really short version

There is no way (except for experience, rules of thumb, and intuition) to know what these parameters should be up front. So, make the product of these two parameters be the size of your available memory:

```
-N 4 -x 4e9
```

for a machine with 16 GB of free memory, for example. Also see the rules of thumb, below.

3.6.2 The short version

These parameters specify the maximum memory usage of the primary data structure in khmer, which is basically N big hash tables of size x. The **product** of the number of hash tables and the size of the hash tables specifies the total amount of memory used.

This table is used to track k-mers. If it is too small, khmer will fail in various ways (and should complain), but there is no harm in making it too large. So, **the absolute safest thing to do is to specify as much memory as is available**. Most scripts will inform you of the total memory usage, and (at the end) will complain if it's too small.

For normalize-by-median, khmer uses one byte per hash entry, so: if you had 16 GB of available RAM, you should specify something like `-N 4 -x 4e9`, which multiplies out to about 16 GB.

For the graph partitioning stuff, khmer uses only 1 bit per k-mer, so you can multiple your available memory by 8: for 16 GB of RAM, you could use

```
-N 4 -x 32e9
```

which multiplies out to 128 Gbits of RAM, or 16 Gbytes.

Life is a bit more complicated than this, however, because some scripts – load-into-counting and load-graph – keep ancillary information that will consume memory beyond this table data structure. So if you run out of memory, decrease the table size.

Also see the rules of thumb, below.

3.6.3 The real full version

khmer's scripts, at their heart, represents k-mers in a very memory efficient way by taking advantage of two data structures, Bloom filters and CountMin Sketches, that are both *probabilistic* and *constant memory*. The "probabilistic" part means that there are false positives: the less memory you use, the more likely it is that khmer will think that k-mers are present when they are not, in fact, present.

Digital normalization (normalize-by-median and filter-abund) uses the CountMin Sketch data structure.

Graph partitioning (load-graph etc.) uses the Bloom filter data structure.

The practical ramifications of this are pretty cool. For example, your digital normalization is guaranteed not to increase in memory utilization, and graph partitioning is estimated to be 10-20x more memory efficient than any other de Bruijn graph representation. And hash tables (which is what Bloom filters and CountMin Sketches use) are really fast and efficient. Moreover, the optimal memory size for these primary data structures is dependent on the number of k-mers, but not explicitly on the size of k itself, which is very unusual.

In exchange for this memory efficiency, however, you gain a certain type of parameter complexity. Unlike your more typical k-mer package (like the Velvet assembler, or Jellyfish or Meryl or Tallymer), you are either guaranteed not to run out of memory (for digital normalization) or much less likely to do so (for partitioning).

The biggest problem with khmer is that there is a minimum hash number and size that you need to specify for a given number of k-mers, and you cannot confidently predict what it is before actually loading in the data. This, by the way, is also true for de Bruijn graph assemblers and all the other k-mer-based software – the final memory usage depends on the total number of k-mers, which in turn depends on the true size of your underlying genomic variation (e.g. genome or transcriptome size), the number of errors, and the k-mer size you choose (the k parameter) [see Conway & Bromage, 2011]. **The number of reads or the size of your data set is only somewhat correlated with the total number of k-mers.** Trimming protocols, sequencing depth, and polymorphism rates are all important factors that affect k-mer count.

The bad news is that we don't have good ways to estimate total k-mer count a priori, although we can give you some rules of thumb, below. In fact, counting the total number of distinct k-mers is a somewhat annoying challenge. Frankly, we recommend *just guessing* instead of trying to be all scientific about it.

The good news is that you can never give khmer too much memory! k-mer counting and set membership simply gets more and more accurate as you feed it more memory. (Although there may be performance hits from memory I/O, e.g. see the NUMA architecture.) The other good news is that khmer can measure the false positive rate and detect dangerously low memory conditions. For partitioning, we actually *know* what a too-high false positive rate is – our k-mer percolation paper lays out the math. For digital normalization, we assume that a false positive rate of 10% is bad. In both cases the data-loading scripts will exit with an error-code.

Rules of thumb

Just use -N 4, always, and vary the -x parameter.

For digital normalization, we recommend:

- `-x 2e9` for any amount of sequencing for a single microbial genome, MDA-amplified or single colony.
- `-x 4e9` for up to a billion mRNAseq reads from any organism. Past that, increase it.
- `-x 8e9` for most eukaryotic genome samples.
- `-x 8e9` will also handle most "simple" metagenomic samples (HMP on down)
- For metagenomic samples that are more complex, such as soil or marine, start as high as possible. For example, we are using `-x 64e9` for ~300 Gbp of soil reads.

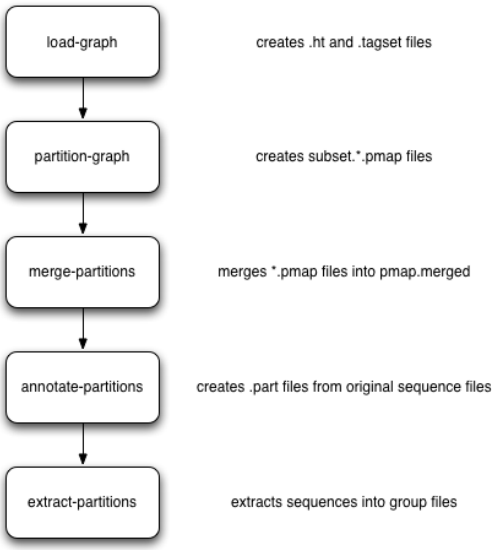
For partitioning of complex metagenome samples, we recommend starting as high as you can – something like half your system memory. So if you have 256 GB of RAM, use `-N 4 -x 256e9` which will use 4 x 256 / 8 = 128 GB of RAM for the basic graph storage, leaving other memory for the ancillary data structures.

3.7 Partitioning large data sets (50m+ reads)

"Partitioning" is what khmer calls the process of separating reads that do not connect to each other into different logical bins. The goal of partitioning is to apply divide & conquer to the process of metagenomic assembly.

3.7.1 Basic partitioning

The basic workflow for partitioning is in the figure below:

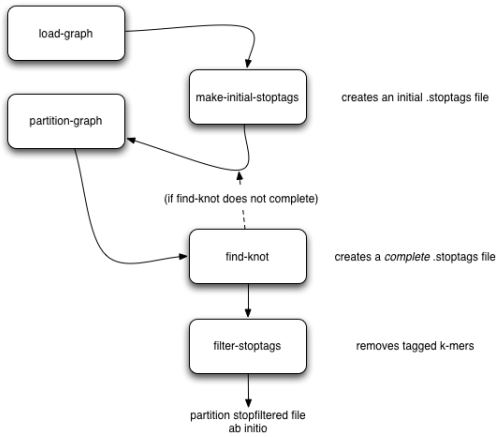


Briefly, you load everything into khmer's probabilistic graph representation; exhaustively explore the graph to find all disconnected sequences; merge the results of the (parallelized) graph exploration; annotate sequences with their partition; and then extract the different partitions into files grouped by partition size. These groups can then be assembled individually.

3.7.2 Artifact removal

As part of our partitioning research, we discovered that large Illumina data sets tend to contain a single large, connected component. This connected component seems to stem from sequencing artifacts that causes knots in the assembly graph. We have developed tools to forcibly remove the knot at the heart of the graph.

Here's the workflow:



3.7.3 Running on an example data set

Here is a set of commands for running both basic partitioning and artifact removal on a small soil metagenomics data set that we've made available for this purpose.

The data set is about 1.1G and you can download it from here:

https://s3.amazonaws.com/public.ged.msu.edu/khmer/iowa-corn-50m.fa.gz

```
cd /path/to/data

# the next command will create a '50m.ct' and a '50m.tagset',
# representing the de Bruijn graph
load-graph.py -k 32 -N 4 -x 16e9 50m iowa-corn-50m.fa.gz

# this will then partition that graph. should take a while.
# update threads to something higher if you have more cores.
# this creates a bunch of files, 50m.subset.*.pmap
partition-graph.py --threads 4 -s 1e5 50m

# now, merge the pmap files into one big pmap file, 50m.pmap.merged
merge-partitions.py 50m

# next, annotate the original sequences with their partition numbers.
# this will create iowa-corn-50m.fa.gz.part
annotate-partitions.py 50m iowa-corn-50m.fa.gz

# now, extract the partitions in groups into 'iowa-corn-50m.groupNNNN.fa'
extract-partitions.py iowa-corn-50m iowa-corn-50m.fa.gz.part

# at this point, you can assemble the group files individually.  Note,
# however, that the last one them is quite big?  this is because it's
# the lump! yay!

# if you want to break up the lump, go through the partitioning bit
# on the group file, but this time with a twist:
mv iowa-corn-50m.group0005.fa corn-50m.lump.fa

# create graph,
load-graph.py -x 8e9 lump corn-50m.lump.fa

# create an initial set of stoptags to help in knot-traversal; otherwise,
```

```
# partitioning and knot-traversal (which is systematic) is really expensive.
make-initial-stoptags.py lump

# now partition the graph, using the stoptags file
partition-graph.py --stoptags lump.stoptags lump

# use the partitioned subsets to find the k-mers that nucleate the lump
find-knots.py -x 2e8 -N 4 lump

# remove those k-mers from the fasta files
filter-stoptags.py *.stoptags corn-50m.lump.fa

# now, reload the filtered data set in and partition again.
# NOTE: 'load-graph.py' uses the file extension to determine
# if the file is formatted as FASTA or FASTQ. The default is
# fasta, therefore if your files are fastq formatted you need
# to append 'fastq' to the name so that 'load-graph.py'
# will parse the file correctly
load-graph.py -x 8e9 lumpfilt corn-50m.lump.fa.stopfilt
partition-graph.py -T 4 lumpfilt
merge-partitions.py lumpfilt
annotate-partitions.py lumpfilt corn-50m.lump.fa.stopfilt
extract-partitions.py corn-50m-lump corn-50m.lump.fa.stopfilt.part

# and voila, after all that, you should now have your de-knotted lump in
# corn-50m-lump.group*.fa.  The *.group????.fa files can now be
# assembled individually by your favorite assembler.
```

3.7.4 Post-partitioning assembly

The 'extract-partitions' script takes reads belonging to each partition and aggregates them into 'group' files; each group file contains at least one entire partition (and generally a lot more). Note, you can control the number of reads in each file (equiv, the size of these files) with some of the arguments that 'extract-partitions' takes.

Now that you have these files... what do you do with them? The short answer is: assemble them! Each of these group files contains reads that do not connect to reads in other files, so the files can be assembled individually (which is the whole point of partitioning).

If you're using Velvet, checkout the `sandbox/velvet-assemble.sh` script, which you can run like this:

```
bash /path/to/khmer/sandbox/velvet-assemble.sh <groupfile> <k>
```

This script does three things:

- first, it breaks the reads up into paired reads and single reads, and puts them in separate files (.pe and .se);
- second, it strips off the partition information from the reads, which confuses Velvet;
- and third, it runs velveth and velvetg to actually assemble.

You can implement your own approach, of course, but this is an example of what we do ourselves.

3.8 Known Issues

Some users have reported that normalize-by-median.py will utilize more memory than it was configured for. This is being investigated in https://github.com/ged-lab/khmer/issues/266

Some FASTQ files confuse our parser when running with more than one thread. For example, while using load-into-counting.py. If you experience this then add "–threads=1" to your command line. This issue is being tracked in https://github.com/ged-lab/khmer/issues/249

If your k-mer table is truncated on write, an error may not be reported; this is being tracked in https://github.com/ged-lab/khmer/issues/443. However, khmer will now (correctly) fail when trying to read a truncated file (See #333).

Paired-end reads from Casava 1.8 currently require renaming for use in normalize-by-median and abund-filter when used in paired mode. The integration of a fix for this is being tracked in https://github.com/ged-lab/khmer/issues/23

Some scripts only output FASTA even if given a FASTQ file. This issue is being tracked in https://github.com/ged-lab/khmer/issues/46

A user reported that abundance-dist-single.py fails with small files and many threads. This issue is being tracked in https://github.com/ged-lab/khmer/issues/75

3.9 An incomplete bibliography of papers using khmer

3.9.1 Digital normalization

Multiple Single-Cell Genomes Provide Insight into Functions of Uncultured Deltaproteobacteria in the Human Oral Cavity. Campbell et al., PLoS One, 2013, doi:10.1371/journal.pone.0059361. [paper link]

Insights into archaeal evolution and symbiosis from the genomes of a nanoarchaeon and its inferred crenarchaeal host from Obsidian Pool, Yellowstone National Park. Podar et al., Biology Direct, 2013 doi:10.1186/1745-6150-8-9. [paper link]

CHAPTER 4

The khmer developer documentation

This section of the documentation is for people who are contributing (or would like to contribute to) the khmer project codebase, either by contributing code or by helping improve the documentation.

Contents:

4.1 Getting started with khmer development

This document is for people who would like to contribute to khmer. It walks first-time contributors through making their own copy of khmer, building it, and submitting changes for review and merge into the master copy of khmer.

Start by making your own copy of khmer and setting yourself up for development; then, build khmer and run the tests; and finally, claim an issue and start developing!

If you're unfamiliar with git and branching in particular, check out the git-scm book.

We've provided a quick guide to the khmer code base here: *A quick guide to the khmer codebase*.

4.1.1 One-time Preparation

1. Install the dependencies.

   OS X users

   (a) Install Xcode from the Mac App Store (requires root).

   (b) Register as an Apple Developer.

   (c) Install the Xcode command-line tools: Xcode -> preferences -> Downloads -> Command Line Tools (requires root).

   Linux users

   (a) Install the python development environment, virtualenv, pip, gcc, and g++.

   On recent Debian and Ubuntu this can be done with:

   ```
   sudo apt-get install python2.7-dev python-virtualenv python-pip gcc \
   g++
   ```

   For RHEL6:

```
sudo yum install -y python-devel python-pip git gcc gcc-c++ make
sudo pip install virtualenv
```

2. Get a GitHub account.

   (We use GitHub to manage khmer contributions.)

3. Fork github.com/ged-lab/khmer.

   Visit that page, and then click on the 'fork' button (upper right).

   (This makes a copy of the khmer source code in your own GitHub account.)

4. Clone your copy of khmer to your local development environment.

   Your clone URL should look something like this:

   ```
   https://github.com/empty-titus/khmer.git
   ```

   and the UNIX shell command should be:

   ```
   git clone https://github.com/empty-titus/khmer.git
   ```

   (This makes a local copy of khmer on your development machine.)

5. Add a git reference to the khmer ged-lab repository:

   ```
   cd khmer
   git remote add ged https://github.com/ged-lab/khmer.git
   cd ../
   ```

   (This makes it easy for you to pull down the latest changes in the main repository.)

6. Create a virtual Python environment within which to work with virtualenv:

   ```
   python2.7 -m virtualenv env
   ```

   This gives you a place to install packages necessary for running khmer.

   OS X users and others may need to download virtualenv first:

   ```
   curl -O https://pypi.python.org/packages/source/v/virtualenv/virtualenv-1.11.6.tar.gz
   tar xzf virtualenv*
   cd virtualenv-*; python2.7 virtualenv.py ../env; cd ..
   ```

   Conda users on any platform can install virtualenv this way:

   ```
   conda install pip
   hash -r
   pip install virtualenv
   python2.7 -m virtualenv env
   ```

7. Activate the virtualenv and install a few packages:

   ```
   source env/bin/activate
   cd khmer
   make install-dependencies
   ```

   (This installs Sphinx and nose, packages we use for building the documentation and running the tests.)

4.1.2 Building khmer and running the tests

1. Activate (or re-activate) the virtualenv:

   ```
   source ../env/bin/activate
   ```

   You can run this many times without any ill effects.

   (This puts you in the development environment.)

2. Build khmer:

   ```
   make
   ```

   If this fails, we apologize – please go create a new issue, paste in the failure message, and we'll try to help you work through it!

   (This takes the C++ source code and compiles it into something that Python can run.)

3. Run the tests:

   ```
   make test
   ```

   You should see lots of output, with something like:

   ```
   Ran 360 tests in 10.403s

   OK
   ```

   at the end.

   (This will run all of the Python tests in the tests/ directory.)

Congratulations! You're ready to develop!

4.1.3 Claiming an issue and starting to develop

1. Find an open issue and claim it.

   Go to the list of open khmer issues and find one you like; we suggest starting with the low-hanging fruit issues).

   Once you've found an issue you like, make sure that no one has been assigned to it (see "assignee", bottom right near "notifications"). Then, add a comment "I am working on this issue." You've staked your claim!

   (We're trying to avoid having multiple people working on the same issue.)

2. In your local copy of the source code, update your master branch from the main khmer master branch:

   ```
   git checkout master
   git pull ged master
   ```

   (This pulls in all of the latest changes from whatever we've been doing on ged-lab.)

3. Create a new branch and link it to your fork on GitHub:

   ```
   git checkout -b fix/issue_number
   git push -u origin fix/issue_number
   ```

   where you replace "issue_number" with the number of the issue.

   (This is the set of changes you're going to ask to be merged into khmer.)

4. Make some changes and commit them.

   This will be issue dependent ;).

   (You should visit and read *Coding guidelines and code review checklist*.)

5. Periodically update your branch from the main khmer master branch:

   ```
   git pull ged master
   ```

   (This pulls in all of the latest changes from whatever we've been doing on ged-lab - important especially during periods of fast change or for long-running pull requests.

6. Run the tests and/or build the docs *before* pushing to GitHub:

   ```
   make doc test pep8
   ```

   Make sure they all pass!

7. Push your branch to your own GitHub fork:

   ```
   git push origin
   ```

   (This pushes all of your changes to your own fork.)

8. Repeat until you're ready to merge your changes into "official" khmer.

9. Set up a Pull Request asking to merge things into the central khmer repository.

   In a Web browser, go to your GitHub fork of khmer, e.g.:

   ```
   https://github.com/empty-titus/khmer
   ```

   and you will see a list of "recently pushed branches" just above the source code listing. On the right side of that should be a "Compare & pull request" green button. Click on it!

   Now:

   - add a descriptive title ("updated tests for XXX")
   - put the issue number in the comment ("fixes issue #532")

   then click "Create pull request."

   (This creates a new issue where we can all discuss your proposed changes; the khmer team will be automatically notified and you will receive e-mail notifications as we add comments. See GitHub flow for more info.)

10. Paste in the committer checklist from *Coding guidelines and code review checklist* and, after its pasted in, check off as many of the boxes as you can.

11. As you add new commits to address bugs or formatting issues, you can keep pushing your changes to the pull request by doing:

    ```
    git push origin
    ```

12. When you are ready to have the pull request reviewed, please add a comment "ready for review!".

13. The khmer team will now review your pull request and communicate with you through the pull request page. Please feel free to add 'ping!' in the comments if you are looking for feedback – this will alert us that you are still on the line – but we will automatically get notified of your pull request and any new comments, so use sparingly.

    If this is still your first issue, please *don't* take another issue until we've merged your first one - thanks!

14. If we request changes, return to the step "Make some changes and commit them" and go from there. Any additional commits you make and push to your branch will automatically be added to the pull request (which is pretty dang cool.)

4.1.4 After your first issue is successfully merged...

You're now an experienced GitHub user! Go ahead and take some more tasks; you can broaden out beyond the low hanging fruit if you like.

Here are a few suggestions:

- If you're knowledgeable in C++ and/or Python and/or documentation and/or biology, we'd love to attract further contributions to khmer. Please visit the issues list and browse about and find something interesting looking.
- One general thing we'd like to do is increase our test coverage. You can go find test coverage information on our continuous integration server by clicking down to individual files; or, ask us on khmer-project@idyll.org for suggestions.
- Ask us! Ask khmer-project@idyll.org for suggestions on what to do next. We can suggest particularly ripe low-hanging fruit, or find some other issues that suit your interests and background.
- You can also help other people out by watching for new issues or looking at pull requests. Remember to be nice and polite!

4.2 A quick guide to testing (for khmer)

This document is for contributors new to automated testing, and explains some of the motivation and logic behind the khmer project's testing approach.

One of our most important "secret sauces" for khmer development is that we do a fair bit of testing to make sure our code works and keeps working!

- We maintain fairly complete test coverage of our code. What this means is that we have automated tests that, when run, execute most of the lines of Python and C++ code in our lib/, khmer/ and scripts/ directories. This doesn't *guarantee* things are correct, but it does mean that at least most of the code works at some basic level.
- we have other tests that we run periodically (for example, before each release) – see *Releasing a new version of khmer* for details. These tests check that our code works on multiple systems and with other people's software.

CTB and others have written a great deal about testing, and testing in Python in particular. Here's an introductory guide CTB wrote a long time ago. You might also be interested in reading this description of the different kinds of tests.

For the more general motivation, see the Lack of Testing Death Spiral.

But... how do you do testing??

First, let's talk about specific goals for testing. What should you be aiming for tests to do? You can always add more testing code, but that might not be useful if they are redundant or over-complicated.

An overall rule is to "keep it simple" – keep things as simple as possible, testing as few things as possible in each test.

We suggest the following approach to writing tests for **new code**:

1. Write a test that just *runs* the new code, generally by copying existing test code to a new test and changing it. Don't do anything clever for the first test – just run something straightforward, and try to use existing data.

2. Decide which use cases should be tested. This is necessarily code specific but our main advice is "don't be clever" – write some tests to make sure that the code basically works.
3. Add in tests for edge cases. By this we mean look for special cases in your code – if statements, fence-post bound errors, etc. – and write tests that exercise those bits of code specifically.

For adding tests to **old code**, we recommend a mix of two approaches:

1. use "stupidity driven testing" and write tests that recapitulate bugs before we fix those bugs.
2. look at test coverage (see khmer's cobertura test coverage, here) and identify lines of C++ or Python code that are not being executed by the current tests. Then write new tests targeting the new code.

Next, to add a test, you have two options: either write a new one from scratch, or copy an existing one. (We recommend the latter.)

To write a new one, you'll need to know how to write tests. For getting an idea of the syntax, read this introductory guide and the official documentation. Then find the right file in `tests/*.py` and add your test!

A better approach is, frankly, to go into the existing test code, find a test that does something similar to what you want to do, copy it, rename it, and then modify it to do the new test.

Finally, *where* do you add new tests and how do you run just *your* test?

Put new tests somewhere in `tests/*.py`. If you have trouble figuring out what file to add them to, just put them in *some* file and we'll help you figure out where to move them when we do code review.

To run one specific test rather than all of them, you can do:

```
./setup.py nosetests --tests tests/test_scripts.py:test_load_into_counting
```

Here, you're running just one test – the test function named `test_load_into_counting` in the file `test_scripts.py`.

4.3 A quick guide to the khmer codebase

This document describes the khmer project's directory layout.

The ChangeLog file lists changes to the codebase, most recent first.

The lib/ directory contains all of the C++ code.

The khmer/ directory contains the khmer package (khmer/__init__.py etc) and the C++-to-Python bridge (khmer/_khmermodule.cc).

The scripts/ and sandbox/ directory contain Python command-line scripts.

The tests/ directory contains all of the tests. Each test is a function in one of the tests/test*.py files.

4.4 Coding guidelines and code review checklist

This document is for anyone who want to contribute code to the khmer project, and describes our coding standards and code review checklist.

4.4.1 Coding standards

All plain-text files should have line widths of 80 characters or less unless that is not supported for the particular file format.

For C++, we use Todd Hoff's coding standard, and astyle -A10 / "One True Brace Style" indentation and bracing. Note: @CTB needs Emacs settings that work for this.

Vim users may want to set the ARTISTIC_STYLE_OPTIONS shell variable to "-A10 –max-code-length=80" and run `:%!astyle` to reformat. The four space indentation can be set with:

```
set expandtab
set shiftwidth=4
set softtabstop=4
```

For Python, PEP 8 is our standard. The `pep8` and `autopep8` Makefile targets are helpful.

Code, scripts, and documentation must have its spelling checked. Vim users can run:

```
:setlocal spell spelllang=en_us
```

Use *]s* and *[s* to navigate between misspellings and *z=* to suggest a correctly spelled word. *zg* will add a word as a good word.

GNU *aspell* can also be used to check the spelling in a single file:

```
aspell check --mode ccpp $filename
```

4.4.2 Code Review

Please read 11 Best Practices for Peer Code Review.

See also Code reviews: the lab meeting for code and the PyCogent coding guidelines.

4.4.3 Checklist

Copy and paste the following into a pull request comment when it is ready for review:

```
- [ ] Is it mergable
- [ ] Did it pass the tests?
- [ ] If it introduces new functionality in scripts/ is it tested?
  Check for code coverage.
- [ ] Is it well formatted? Look at `pep8`/`pylint`, `cppcheck`, and
  `make doc` output. Use `autopep8` and `astyle -A10 --max-code-length=80`
  if needed.
- [ ] Is it documented in the ChangeLog?
- [ ] Was a spellchecker run on the source code and documentation after changes
  were made?
```

Note that after you submit the comment you can check and uncheck the individual boxes on the formatted comment; no need to put x or y in the middle.

4.5 A guide for khmer committers

This document is for people with commit rights to github.com/ged-lab/khmer.

If you have commit privileges to the ged-lab/khmer repository, here are a few useful tips.

First, never merge something unless it's been through a review! This rule can be broken under specific conditions when doing a release; see *Releasing a new version of khmer*.

Second, need to force another continuous integration run? Put "test this please" in a comment. This can be used to ask our continuous integration system to run on someone else's pull request – by default, it only runs on commits from people who have write privileges to khmer, so you may need to do this if you're reviewing someone else's pull request.

Third, we ask that all contributors set up standing Pull Requests while they are working something. (This is a **requirement** if you're in the GED lab.) This lets us track what's going on. On the flip side, please do not review pull requests until they are indicated as "ready for review".

4.6 Releasing a new version of khmer

This document is for khmer release managers, and details the process for making a new release of the khmer project.

4.6.1 How to make a khmer release candidate

Michael R. Crusoe, Luiz Irber, and C. Titus Brown have all been release makers, following this checklist by MRC.

1. The below should be done in a clean checkout:

   ```
   cd `mktemp -d`
   git clone git@github.com:ged-lab/khmer.git
   cd khmer
   ```

2. (Optional) Check for updates to versioneer:

   ```
   pip install versioneer
   versioneer-installer

   git diff

   ./setup.py versioneer
   git diff
   git commit -m -a "new version of versioneer.py"
   # or
   git checkout -- versioneer.py khmer/_version.py khmer/__init__.py MANIFEST.in
   ```

3. Review the git logs since the last release and diffs (if needed) and ensure that the `ChangeLog` is up to date:

   ```
   git log --minimal --patch `git describe --tags --always --abbrev=0`..HEAD
   ```

4. Review the issue list for any new bugs that will not be fixed in this release. Add them to `doc/known-issues.txt`

5. Verify that the build is clean: http://ci.ged.msu.edu/job/khmer-master/

6. Submit a build to Coverity Scan if it hasn't been done recently. You can get the token from https://gitlab.msu.edu/ged-lab/ged-internal-docs/wikis/coverity-scan or https://scan.coverity.com/projects/621?tab=project_settings

   ```
   make clean
   cov_analysis_dir=~/src/coverity/cov-analysis-linux64-7.0.2/ make coverity-build
   COVERITY_TOKEN=${COVERITY_TOKEN} make coverity-upload
   ```

7. Set your new version number and release candidate:

```
new_version=1.1
rc=rc3
```

and then tag the release candidate with the new version number prefixed by the letter 'v':

```
git tag v${new_version}-${rc}
git push --tags git@github.com:ged-lab/khmer.git
```

8. Test the release candidate. Bonus: repeat on Mac OS X:

```
cd ..
virtualenv testenv1
virtualenv testenv2
virtualenv testenv3
virtualenv testenv4
# First we test the tag

cd testenv1
source bin/activate
git clone --depth 1 --branch v${new_version}-${rc} https://github.com/ged-lab/khmer.git
cd khmer
make install
nosetests khmer --attr '!known_failing'
make test
normalize-by-median.py --version # double-check version number

# Secondly we test via pip

cd ../../testenv2
source bin/activate
pip install -U setuptools==3.4.1
pip install -e git+https://github.com/ged-lab/khmer.git@v${new_version}-${rc}#egg=khmer
cd src/khmer
make dist
make install
nosetests khmer --attr '!known_failing'
make test
normalize-by-median.py --version # double-check version number
cp dist/khmer*tar.gz ../../../testenv3/

# Is the distribution in testenv2 complete enough to build another
# functional distribution?

cd ../../../testenv3/
source bin/activate
pip install -U setuptools==3.4.1
pip install khmer*tar.gz
nosetests khmer --attr '!known_failing'
tar xzf khmer*tar.gz
cd khmer*
make dist
make test
```

9. Publish the new release on the testing PyPI server. You will need to change your PyPI credentials as documented here: https://wiki.python.org/moin/TestPyPI. You may need to re-register:

```
python setup.py register --repository test
```

Now, upload the new release:

```
python setup.py sdist upload -r test
```

Test the PyPI release in a new virtualenv:

```
cd ../../testenv4
source bin/activate
pip install -U setuptools==3.4.1
pip install screed nose
pip install -i https://testpypi.python.org/pypi --pre --no-clean khmer
nosetests khmer --attr '!known_failing'
normalize-by-median.py --version 2>&1 | awk ' { print $2 } '
cd build/khmer
./setup.py nosetests
```

10. Do any final testing (BaTLab and/or acceptance tests).
11. Make sure any release notes are merged into doc/release-notes/.

4.6.2 How to make a final release

When you've got a thoroughly tested release candidate, cut a release like so:

1. Create the final tag and publish the new release on PyPI (requires an authorized account).:

```
cd ../../../khmer
git tag v${new_version}
python setup.py register sdist upload
```

2. Delete the release candidate tag and push the tag updates to GitHub.:

```
git tag -d v${new_version}-${rc}
git push git@github.com:ged-lab/khmer.git
git push --tags git@github.com:ged-lab/khmer.git
```

3. Add the release on GitHub, using the tag you just pushed. Name it 'version X.Y.Z', and copy and paste in the release notes.
4. Make a binary wheel on OS X.:

```
virtualenv build
cd build
source bin/activate
pip install -U setuptools==3.4.1 wheel
pip install --no-clean khmer==${new_version}
cd build/khmer
./setup.py bdist_wheel upload
```

5. Update Read the Docs to point to the new version. Visit https://readthedocs.org/builds/khmer/ and 'Build Version: master' to pick up the new tag. Once that build has finished check the "Activate" box next to the new version at https://readthedocs.org/dashboard/khmer/versions/ under "Choose Active Versions". Finally change the default version at https://readthedocs.org/dashboard/khmer/advanced/ to the new version.
6. Delete any RC tags created:

```
git tag -d ${new_version}-${rc}
git push origin :refs/tags/${new_version}-${rc}
```

7. Tweet about the new release.
8. Send email including the release notes to khmer@lists.idyll.org and khmer-announce@lists.idyll.org

4.6.3 BaTLab testing

The UW-Madison Build and Test Lab provides the khmer project with a free cross-platform testing environment.

1. Connect to their head node:

   ```
   ssh mcrusoe@submit-1.batlab.org
   ```

2. Move into the khmer directory and download a release from PyPI's main server or the test PyPI server:

   ```
   cd khmer/
   wget https://testpypi.python.org/packages/source/k/khmer/khmer-1.0.1-rc3.tar.gz
   vim khmer-v1.0.inputs # change the 'scp_file' to point to the release
   vim khmer-v1.0.run-spec # change 'project_version' at bottom
   nmi_submit khmer-v1.0.run-spec
   ```

4.6.4 Setuptools Bootstrap

ez_setup.py is from https://bitbucket.org/pypa/setuptools/raw/bootstrap/

Before major releases it should be examined to see if there are new versions available and if the change would be useful

4.6.5 Versioning Explanation

Versioneer, from https://github.com/warner/python-versioneer, is used to determine the version number and is called by Setuptools and Sphinx. See the files `versioneer.py`, the top of `khmer/__init__.py`, `khmer/_version.py`, `setup.py`, and `doc/conf.py` for the implementation.

The version number is determined through several methods: see https://github.com/warner/python-versioneer#version-identifiers

If the source tree is from a git checkout then the version number is derived by `git describe --tags --dirty --always`. This will be in the format `${tagVersion}-${commits_ahead}-${revision_id}-${isDirty}`. Example: `v0.6.1-18-g8a9e430-dirty`

If from an unpacked tarball then the name of the directory is queried.

Lacking either of the two git-archive will record the version number at the top of `khmer/_version.py` via the `$Format:%d$` and `$Format:%H$` placeholders enabled by the "export-subst" entry in `.gitattributes`.

Non source distributions will have a customized `khmer/_version.py` that contains hard-coded version strings. (see `build/*/khmer/_version.py` after a `python setup.py build` for an example)

`ez_setup.py` bootstraps Setuptools (if needed) by downloading and installing an appropriate version

4.7 Miscellaneous implementation details

Partition IDs are "stored" in FASTA files as an integer in the last tab-separated field. Yeah, dumb, huh?

4.8 Development miscellany

4.8.1 Third-party use

We ask that third parties who build upon the codebase to do so from a versioned release. This will help them determine when bug fixes apply and generally make it easier to collaborate. If more intensive modifications happen then we request that the repository is forked, again preferably from a version tag.

4.8.2 Build framework

'make' should build everything, including tests and "development" code.

4.8.3 git and GitHub strategies

Still in the works, but read this.

Make a branch on ged-lab (preferred so others can contribute) or fork the repository and make a branch there.

Each piece or fix you are working on should have its own branch; make a pull- request to ged-lab/master to aid in code review, testing, and feedback.

If you want your code integrated then it needs to be mergable

Example pull request update using the command line:

1. **Clone the source of the pull request (if needed)** `git clone git@github.com:mr-c/khmer.git`
2. **Checkout the source branch of the pull request** `git checkout my-pull-request`
3. **Pull in the destination of the pull request and resolve any conflicts** `git pull git@github.com:ged-lab/khmer.git master`
4. Push your update to the source of the pull request `git push`
5. Jenkins will automatically attempt to build and test your pull requests.

4.8.4 Code coverage

Jenkins calculates code coverage for every build. Navigate to the results from the master node first to view the coverage information.

Code coverage should never go down and new functionality needs to be tested.

4.8.5 Pipelines

All khmer scripts used by a published recommended analysis pipeline must be included in scripts/ and meet the standards therein implied.

4.8.6 Command line scripts

Python command-line scripts should use '-' instead of '_' in the name. (Only filenames containing code for import imported should use _.)

Please follow the command-line conventions used under scripts/. This includes most especially standardization of '-x' to be hash table size, '-N' to be number of hash tables, and '-k' to always refer to the k-mer size.

Command line thoughts:

> If a filename is required, typically UNIX commands don't use a flag to specify it.
>
> Also, positional arguments typically aren't used with multiple files.
>
> CTB's overall philosophy is that new files, with new names, should be created as the result of filtering etc.; this allows easy chaining of commands. We're thinking about how best to allow override of this, e.g.
>
> ```
> filter-abund.py <ct file> <filename> [ -o <filename.keep> ]
> ```

All code in scripts/ must have automated tests; see tests/test_scripts.py. Otherwise it belongs in sandbox/.

When files are overwritten, they should only be opened to be overwritten after the input files have been shown to exist. That prevents stupid command like mistakes from trashing important files.

It would be nice to allow piping from one command to another where possible. But this seems complicated.

CTB: should we squash output files (overwrite them if they exist), or not? So far, leaning towards 'not', as that way no one is surprised and loses their data.

A general error should be signaled by exit code *1* and success by *0*. Linux supports exit codes from *0* to *255* where the value *1* means a general error. An exit code of *-1* will get converted to *255*.

CLI reading:

> http://stackoverflow.com/questions/1183876/what-are-the-best-practices-for-implementing-a-cli-tool-in-perl
>
> http://catb.org/esr/writings/taoup/html/ch11s06.html
>
> http://figshare.com/articles/tutorial_pdf/643388

4.8.7 Python / C integration

The Python extension that wraps the C++ core of khmer lives in khmer/_khmermodule.CC

This wrapper code is tedious and annoying so we use a static analysis tool to check for correctness.

https://gcc-python-plugin.readthedocs.org/en/latest/cpychecker.html

Developers using Ubuntu Precise will want to install the gcc-4.6-plugin-dev package

Example usage:

```
CC="/home/mcrusoe/src/gcc-plugin-python/gcc-python-plugin/gcc-with-cpychecker
--maxtrans=512" python setup.py build_ext 2>&1 | less
```

False positives abound: ignore errors about the C++ standard library. This tool is primarily useful for reference count checking, error-handling checking, and format string checking.

Errors to ignore: "Unhandled Python exception raised calling 'execute' method", "AttributeError: 'NoneType' object has no attribute 'file"'

Warnings to address:

```
khmer/_khmermodule.cc:3109:1: note: this function is too complicated
for the reference-count checker to fully analyze: not all paths were
analyzed
```

Adjust –maxtrans and re-run.

```
khmer/_khmermodule.cc:2191:61: warning: Mismatching type in call to
Py_BuildValue with format code "i" [enabled by default]
  argument 2 ("D.68937") had type
    "long long unsigned int"
  but was expecting
    "int"
  for format code "i"
```

See below for a format string cheat sheet One also benefits by matching C type with the function signature used later.

“I” for unsigned int “K” for unsigned long long a.k.a khmer::HashIntoType.

4.9 Deploying the khmer project tools on Galaxy

This document is for people interested in deploying the khmer tools on the Galaxy platform.

We are developing the support for running normalize-by-median in Galaxy.

When this is mature we will make a Galaxy Tool Shed version available for easier installation.

4.9.1 Install the tools & tool description

If your installation uses a virtualenv be sure to activate it in your terminal before continuing.

```
pip install --no-clean khmer
```

Move to the *tools* directory in your Galaxy installation and copy in the tool definition file.:

```
cd tools
mkdir khmer
ln -s build/khmer/scripts/normalize-by-median.xml .
```

Add the following to your tool_conf.xml inside the <toolbox> tag:

```
<section id="khmer-protocols-extra" name="khmer protocols">
<tool file="khmer/normalize-by-median.xml" />
</section>
```

Then (re)start Galaxy.

4.9.2 Single Output Usage

For one or more files into a single file:

#. Choose ‘Normalize By Median’ from the ‘khmer protocols’ section of the ‘Tools’ menu.

#. Compatible files already uploaded to your Galaxy instance should be listed. If not then you may need to set their datatype manually.

#. After selecting the input files specify if they are paired-interleaved or not.

#. Specify the sample type or show the advanced parameters to set the tablesize yourself. Consult *Choosing table sizes for khmer* for assistance.

4.10 Crazy ideas

1. A JavaScript preprocessor to do things like count k-mers (HLL), and do diginorm on data as uploaded to server.

   Inspired by a paper that Titus reviewed for PLoS One; not yet published.

CHAPTER 5

License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the Michigan State University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.